A Unifying Theory and Improvements for Existing Approaches to Text Compression

---

A thesis

submitted in partial fulfilment

of the requirements for the Degree of

Doctor of Philosophy in Computer Science

in the

University of Canterbury

by

Timothy Bell

University of Canterbury

1986

Contents

Abstract

More than 40 different schemes for performing text compression have been proposed in the literature. Many of these schemes appear to use quite different approaches, such as Huffman coding, dictionary substitution, predictive modelling, and modelling with Finite State Automata (FSA). From the many schemes in the literature, a representative sample has been selected to include all schemes of current interest (i.e. schemes which are in popular use, or those which have been proposed recently). The main result given in the thesis is that each of these schemes disguises some form of *variable-order Markov model* (VOMM), which is a relatively inexact model for text.

In a variable-order Markov model, each symbol is predicted using a finite number of directly preceding symbols as a context. An important class of FSAs, called Finite Context Automata (FCAs) is defined, and is shown that FCAs implement a form of variable-order Markov modelling. Informally, an FCA is an FSA where the current state is determined by some finite number of immediately preceding input symbols.

Three types of proof are used to show that text compression schemes use variable-order Markov modelling: (1) some schemes, such as Cleary and Witten's "Prediction by Partial Matching", use a VOMM by definition, (2) Cormack and Horspool's "Dynamic Markov Compression" scheme uses an FSA for prediction, and it is shown that the FSAs generated will always be FCAs, (3) a class of compression schemes called Greedy Macro (GM) schemes is defined, and a wide range of compression schemes, including Ziv-Lempel (LZ) coding, are shown to belong to that class. A construction is then given to generate an FSA equivalent to any GM scheme, and the FSA is shown to implement a form of variable-order Markov modelling.

Because variable-order Markov models are only a crude model for text, the main conclusion of the thesis is that more powerful models, such as Pushdown Automata,

combined with arithmetic coding, offer better compression than any existing schemes, and should be explored further. However, there is room for improvement in the compression and speed of some existing schemes, and this is explored as follows.

The LZ schemes are currently regarded as the most practical, in that they achieve good compression, are usually very fast, and require relatively little memory to perform well. To study these schemes more closely, an explicit probabalistic *symbol-wise* model is given, which is equivalent to one of the LZ schemes, LZ77. This model is suitable for providing probabilities for character-by-character Huffman or arithmetic coding. Using the insight gained by examining the symbol-wise model, improvements have been found which can be reflected in LZ schemes, resulting in a scheme called LZB, which offers improved compression, and for which the choice of parameters is less critical. Experiments verify that LZB gives better compression than competing LZ schemes for a large number of texts.

Although the time complexity for encoding using LZB and similar schemes is $O(n)$ for a text of n characters, straightforward implementations are very slow. The time consuming step of these algorithms is a search for the longest string match. An algorithm is given which uses a binary search tree to find the longest string match, and experiments show that this results in a dramatic increase in encoding speed.

"There is nothing new under the sun.
Is there anything of which one can say,
'Look! There is something new'?
It was here already, long ago..."

Ecclesiastes 1:9,10

Chapter 1
Introduction

1.1 Text compression without computers

Towards the end of the eighteenth century the British Admiralty needed a fast means of sending messages between London and the naval stations on the coast. A system was devised using a series of cabins, typically about 5 miles apart, on hilltops between London and the ports. Each cabin had six large *shutters* on its roof (Figure 1.1), which could be seen from adjacent cabins. A message was transmitted along the "line" by setting up a pattern in a set of shutters in London, which was relayed by operators at each cabin, until it reached the port. These *shutter telegraphs* were capable of transmitting messages over many miles in just a few minutes [Wilson 76].

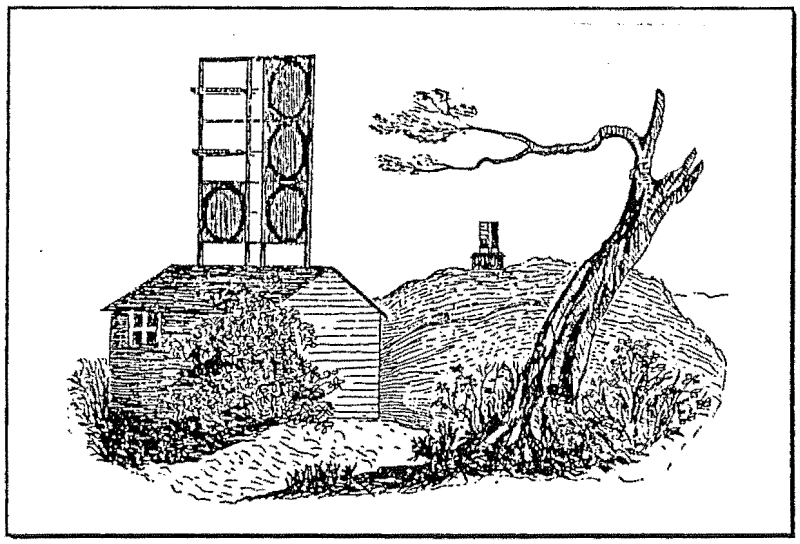


Figure 1.1: A shutter telegraph cabin

Different combinations of shutters represented the letters of the alphabet and the numerals. The six shutters had 64 combinations, so some combinations were spare. These spare combinations were used to represent common words and phrases such as "and",

"the", "Portsmouth", "West", and even "Sentence of court-martial to be put into execution". This *codebook* approach enabled messages to be transmitted considerably faster than could be done with a letter-by-letter approach. The codebook shutter telegraph system is an early example of *text compression*; that is, the use of short codes for common messages and longer codes for uncommon messages, resulting in a reduction in the length of the average message. There were two main drawbacks to the codebook scheme. One was that the operators at the terminating stations had to be able to use a codebook, and this skill commanded a higher wage. The second was that although few errors occurred, the effect of one error was considerably greater when a codebook was used, than for a letter-by-letter transmission. Consider the effect of the erroneous receipt of the code for "Sentence of court-martial to be put into execution"!

The advantages and disadvantages of using a codebook illustrate some of the main issues about the use of text compression: the tradeoff of faster transmission (or less storage space), against more effort for reading and writing, and the problem of errors.

Another small advantage of the codebook scheme was that a person observing a relay station would have considerable difficulty interpreting a coded message, and so some security was achieved for the information being transmitted.

In the 1820s, Louis Braille devised a system, still in common use today, which enables the blind to "read" by touch. In the Braille system (Figure 1.2), text is represented on a thick sheet of paper by raised *dots*. The dots are grouped in *cells* of six, with each dot either flat or raised. Each cell usually represents a letter of the alphabet. As with the shutter telegraphs, not all of the 64 possible combinations are needed to represent the alphabet and numerals, and the spare codes are used to represent common words ("and", "for", "of", etc.) and common groups of letters ("ch", "gh", "sh", etc.). More sophisticated Braille readers also use other codes, such as *short form words*, which are made up of two cells. The first cell indicates that a short form word follows, and the second cell is an ordinary letter code which represents the word.

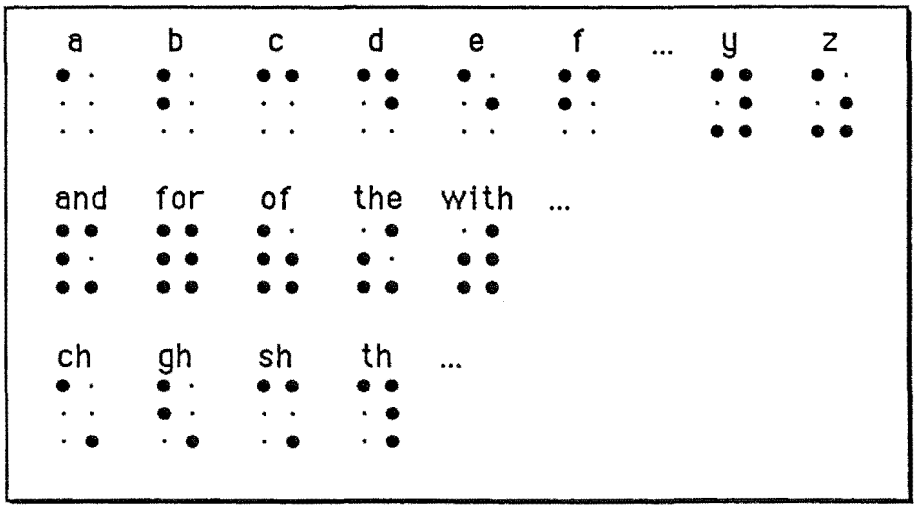


Figure 1.2: The Braille system

Using these codes, the space needed to represent a text is reduced significantly. Also, usually the slowest part of reading Braille is recognising the cells. Because a compressed text contains fewer cells it can be read faster, at the expense of some extra effort required of the reader to interpret the codes. In Figure 1.3, 21 characters are coded in 9 cells, illustrating the amount of compression that can be achieved.

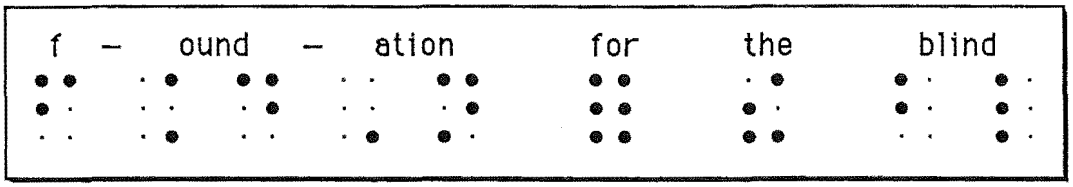


Figure 1.3: A compressed Braille coding

In 1838 Morse invented his well known code which is used for transmitting messages by electric telegraphs, flashing lights, and other media. In Morse code each character is represented as a series of dots and dashes (Figure 1.4). Common characters are allocated short codes and less common characters have longer codes. This reduces the average time taken for messages, and is another form of text compression.

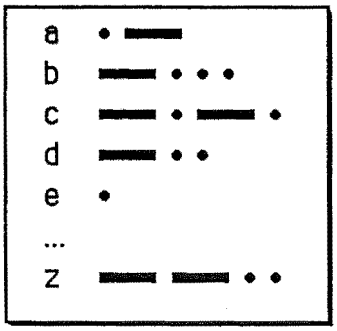


Figure 1.4: Morse code

In his *Human behaviour and the principle of least effort*, Zipf [Zipf 49] points out that the length of words in the English language are the result of our natural attempt to reduce the average time taken to communicate. Common words like "of", "and", and "a" are short, while less frequently used words, such as "compression" tend to be longer. We have seen this shortening process happen in our own lifetimes with words like "microprocessor" being changed to "micro", and even "μp"! Our language also includes other compression devices, such as acronyms ("Nato", "BBC") and abbreviations ("abbrev.", "etc.").

1.2 Text compression

1.2.1 Definition

A *Text Compression (TC) scheme* is a pair of algorithms: an encoder, which performs compression, and a decoder, which performs decompression. The encoder converts a text to a compressed form. The decoder takes the compressed form and reconstructs the original text *exactly*. For the TC scheme to be useful, the size of the compressed form must be less than the size of the original text. It is not unusual for a TC scheme to reduce text to around a half of its original size, and more sophisticated approaches can reduce a text down to a third of its original size.

1.2.2 Text

For the purposes of this thesis, *text* will be defined to be the subset of data which can be represented on a computer using a standard character set such as ASCII or EBCDIC. This includes English text, source code, on-line manuals and data transmitted to terminals. Other types of data, such as pictures, machine code, and music, are excluded because they are best compressed by their own specialised methods. The schemes here will be considered both theoretically and empirically in the light of the structure of languages, both natural and artificial. Note that some of the TC schemes described can be applied to a wider range of data than just text.

1.2.3 Compression and compaction

The terms *compression* and *compaction* have appeared in the literature with a range of meanings. The definitions used here are from [Gottlieb 75].

Compaction of data means any technique which reduces the size of the physical representation of the data while preserving a subset of the information deemed "relevant information".

Compression of data is a compaction technique which is completely reversible.

Compaction includes techniques such as removing multiple blanks, abbreviation [Bourne 61], and file deletion! Only compression schemes are to be considered here.

1.2.4 Adaptive coding

Adaptive coding is an important technique in text compression, used by several schemes. It is illustrated here by applying it to the *shutter telegraphs* of section 1.1.

Recall that the operators at each end of the telegraph were given identical codebooks. If the operators were reliable, all messages would be correctly encoded and decoded. Let us now suppose that the Navy's telegraph system is open to the general public. There would

now be a wide variety of topics being transmitted through the telegraph system, and the codebooks used would not be very helpful. For example, there would be little demand for messages like "Sentence of court-martial to be put into execution".

The supervisor of the telegraph would probably have a new codebook prepared containing more general common words, and issue a copy to each operator. To get a copy of the codebook to the distant operator, he could have it sent through the telegraph. If the supervisor wanted the codebooks to be even better suited to the messages being sent, he could require the operators to use the following *semi-adaptive* scheme.

Before the operator sends a message, he reads it, writes down the most frequently used words, and uses them to make up a codebook specifically for that message. He then transmits the codebook, followed by the coded message. The receiving operator writes down the codebook, and then uses it to decode the message.

A disadvantage of the semi-adaptive scheme is that the sending operator must see the entire message before it can be transmitted. Suppose there is a need for a special "urgent" service, where the message is transmitted as it is dictated, so that each word is repeated at the receiving station only a few seconds after it is spoken at the transmitting station. This urgent service could be performed using the following *adaptive* scheme.

The sending operator begins transmitting a message letter by letter, which the receiving operator can decode. Both operators write down each word as it is transmitted. By agreement, when a word has been transmitted or received twice in the same text, both operators add it to the codebook, allocating it the next spare code. From there on that word can be coded using the new code. Before long the codebooks will contain codes specifically tailored for the topic and language of the message. Of course, the system assumes that the operators are completely reliable and consistent.

The illustration shows how remarkable adaptive coding is. Both operators have identical codebooks well suited to the message, yet *the codes were never explicitly*

transmitted. In general a compression scheme is said to use adaptive coding if the code used for a particular character (or phrase) is based only on the text already transmitted.

The adaptive shutter telegraph system might not work very well in practice due to the operators making mistakes in the construction of the codebook. However, this is not a problem with computers, as the encoder and decoder can be made religiously consistent by using an exact duplicate of the encoder's codebook construction algorithm in the decoder.

Cleary and Witten [Cleary 84a] explore adaptive coding, and show that it compares very favourably with non-adaptive approaches, and yet permits single pass coding.

1.3 Advantages and disadvantages of text compression

Many of the advantages and disadvantages of TC have been illustrated by the non-computer applications. The best known advantages are the saving of storage space and the decrease in time taken to transmit data. Often a system's performance will increase with the appropriate use of compression. For example, on the UUCP network, it has been shown that the number of CPU cycles used by applying a compression scheme to files before transmission is more than made up for by the saving of CPU cycles during transmission. In addition, the transmission takes less time, and consequently costs less [Fair 86].

TC also allows easier distribution of decentralised data [Urrows 84]. Compression is even being applied to mass storage devices, such as CD ROM, because of the demand to store more and more information [Lambert 86].

Another advantage of TC is that the smaller size of compressed files will encourage more comprehensive backups and archives, since the backup time is reduced, and fewer tapes and shelves are needed to store backups.

Because TC removes redundancy from a text file, it has other useful side effects. Some amount of encryption can be achieved because the lack of redundancy removes the opportunity to use statistical regularities to decipher a code. The "randomness" of a

compressed text may also be desirable for error detection, because redundancy can be added in a controlled way to a compressed text to combat particular types of noise (errors).

Finally, fast searching can be achieved with some TC techniques by compressing the search key, and searching the shorter compressed text.

One of the disadvantages of compression is the extra CPU time required for encoding and decoding. However, it has already been seen that it is not unusual for the extra CPU time to be more than made up for by other savings.

Another objection to the use of compression is the disastrous effect that a small error can have. Although this is the case, it must be remembered that there will be less data for the error to occur in and so the probability of *no* errors occurring is greater for compressed text than uncompressed text! And if an error does occur, it will be very obvious to the user; in an uncompressed text the error might cause a single important character to silently change, whereas an error in a compressed text can result in pages of garbage.

Implementing random access to the records of a file is complicated considerably by compressing the file, because fixed-length records cannot easily be used.

Another drawback with the use of compression is the lack of standards, and the difficulty of implementation. This situation has been changing recently with the widespread use of electronic mail, and the free distribution of the high-performance general purpose compression program "compress" [Thomas 85], which is available for several different computers.

The study of TC is important in the field of Computer Science because it involves the practical application of information theory and the AI techniques of modelling and prediction. The amount of compression achieved can be used as a measure of how "good" a model is for a language.

1.4 Definitions

Some important terms are introduced here. A more comprehensive list of terms, including the names of TC schemes, appears in the Glossary.

A *character* is any member of a character set such as ASCII or EBCDIC. It is a fundamental component of text. An *alphabet* is the set of all possible characters which can occur in a text, and is usually denoted as A. A *string* $s=s_1s_2\cdots s_n$ is a finite sequence of n characters. Its length may also be denoted as |s|. If |s|=0 then it is the *empty string*, denoted by Λ. In the literature the terms *string*, *text* and *file* are often used interchangeably to mean "the target of a TC scheme", and no distinction will be made here. The term *symbol* usually refers to a character, but may also be some other easily recognised type of string, such as an English word.

The task of designing a text compression scheme has been split into two parts: *modelling* and *coding* [Rissanen 81]. With the shutter telegraphs, the *model* was the codebook, and the *code* was the different combinations of shutters. In general a *model* attempts to predict (allocate probabilities to) the next symbol to be encoded, although the model might be very crude. The *code* is designed to transmit the symbol being encoded by transmitting the behaviour of the model when that symbol is encountered. In order to make transmission as efficient as possible, symbols which are given a high probability by the model are usually allocated short codes, and vice versa. A variety of approaches to modelling and coding are discussed in subsequent chapters. Sometimes the model and code used by a scheme are not immediately obvious, and much of the work presented here will reveal the model and code for schemes where they are obscure.

A model provides a probability distribution for the encoder in which the probability of each possible symbol is estimated. The estimated probability of a character c is denoted by the function P(c). The upper case P distinguishes the estimated probability from the true probability, p(c). The estimated probability of a symbol is sometimes referred to as the *code space* allocated to that symbol. For example, if the character 'e' has an estimated

probability P('e')=1/4, then we say it is allocated 1/4 of the code space. If P('e')=1 it is allocated *all* of the code space, and if P('e')=0, it is allocated *none* of the code space.

1.5 Outline of the thesis

The main theme of this thesis is that every TC scheme in the literature belongs to a class of models which is relatively inexact, called variable-order Markov models. The conclusion drawn from this is that more powerful models, such as Pushdown automata, should be explored to achieve better compression. The theme is developed as follows.

In chapter 2, a wide range of TC schemes is described in detail, and evaluated empirically. In chapter 3, the variable-order Markov model (VOMM) is defined, and TC schemes which are obviously based on that paradigm are identified. A subset of Finite State Automata, called Finite Context Automata (FCAs) is also defined, and it is shown that a scheme using an FCA is a VOMM. In chapter 4 the DMC TC scheme is shown to use an FCA as its model, and so uses variable-order Markov modelling. Chapter 5 collects together the remaining TC schemes, including the well-known Ziv-Lempel schemes, in a class called Greedy Macro (GM) schemes. A general procedure is given for decomposing any GM scheme to a VOMM.

The empirical results of chapter 2 show that the Ziv-Lempel (LZ) compression schemes offer the best compression of any schemes currently used in practice. A secondary subject of the thesis is the exploration of improvements for LZ coding. A specific VOMM TC scheme equivalent to the LZ77 scheme is given in chapter 6. This equivalent scheme reveals inefficiencies in LZ77 coding, some of which can be changed in the original, resulting in the improved compression scheme, LZB. One of the main disadvantages of LZB and similar Ziv-Lempel schemes is that encoding can be very slow. In chapter 7 an algorithm is given to speed up LZ encoding, making LZB a more practical scheme.

Chapter 2
Existing Compression Schemes

2.1 History

In 1951, Claude Shannon published his *Prediction and Entropy of Printed English* [Shannon 51] suggesting that because some redundancy was present in English text, it could be represented compactly on computers if suitable encoding was used. A year later, David Huffman published a *Method for the Construction of Minimum-Redundancy Codes* [Huffman 52], and it seemed that the optimal compression scheme had been found. Indeed, many people still consider that Huffman coding is optimal [McIntyre 85], and are not aware that more than 40 approaches to compression have been published since Huffman's contribution, many of them offering significantly better performance [Horspool 86]. Even recent surveys concentrate on Huffman coding [Severance 83, Bassiouni 85].

Further work on Huffman codes led to variations with special properties, such as preserving alphabetical order, decreasing the effect of channel errors and obtaining better compression by coding blocks of characters [Gilbert 59, Schwartz 64, Hu 71, Ruth 72]. More recently, algorithms have been developed which allow Huffman coding to be used adaptively [Gallager 78, Cormack 84, Knuth 85].

Various *ad hoc* methods have been published, each of which disguise a form of variable length coding which is an approximation to Huffman coding [Lynch 73, Hahn 74, Lea 78].

The compression schemes which perform the best have generally been published since the late 1970s, because the availability of fast CPUs and large memories has allowed sophisticated schemes to be implemented.

Before this, several approaches were used. Dictionary coding [White 67] is where words in a text are replaced with indexes to a dictionary. Some improvements to this were found by Young and Liu [Young 80]. Digram coding is a fast and simple method which appeared in the early 1970s [Snyderman 70, Schieber 71, Jewell 76, Bookstein 76]. In the mid 1970s, much work was done on parsing techniques, where the encoder searched for common phrases, and used these as a dictionary for encoding [Schuegraf 73, Wagner 73, Schuegraf 74, Mayne 75, Rubin 76, Wolff 78].

In the late 1970s, two quite different approaches were discovered, which are still the subject of most current research in compression, and represent the state of the art.

One of these innovations is Ziv-Lempel (LZ) coding [Ziv 77, Ziv 78], which is a development of earlier parsing techniques, with the innovative idea of using a text as its own dictionary. In LZ coding, a phrase (group of characters) is replaced by a pointer to a previous occurrence of that phrase in the text. Further work in LZ coding is found in [Rodeh 81, Storer 82, Welch 84, Jakobsson 85, Thomas 85, Bell 86].

The other innovation is arithmetic coding [Pascoe 76, Rissanen 76, Rissanen 79, Guazzo 80, Jones 81, Langdon 81, Witten 86], which is a generalisation of Huffman coding, without the restriction that each input symbol is represented by an integer number of bits. This allows skew probability distributions to be coded effectively, and led to schemes which use a large context to predict each character in a text [Langdon 83b, Cleary 84b, Cormack 87].

Because so many different compression schemes have been proposed, an exhaustive evaluation would be a major task. However, many of the schemes have been superceded, particularly with the discovery of LZ and arithmetic coding. On the assumption that any of the older schemes which are of practical use would still be prominent, the schemes chosen for evaluation are those which are in current use, or have been recently published, including all known schemes which use LZ or arithmetic coding.

2.2 Evaluating the performance of text compression schemes

Nineteen approaches to TC are evaluated in this chapter. The most common measure of the performance of a TC scheme is the amount of compression achieved, but other important factors are the speed and memory required for encoding and for decoding, and the difficulty of implementation.

An empirical comparison for the speed and memory usage of the schemes was not possible because some of the implementations evaluated only the amount of compression achieved, rather than actually performing the compression. Instead, an indication of the resources required for each scheme will be given with its description.

The amount of compression achieved by each scheme has been measured for six benchmark texts, and the results are presented at the end of this chapter in Table 2.2. A few lines from each text are shown in Figure 2.1. The benchmark texts were chosen to represent a variety of types of text. They are:

(1) matthew: a book from the Good News Bible (139,521 characters)

(2) short: the first 100 lines of matthew (4,510 characters)

(3) csh: an online manual on a UNIX system (60,997 characters)

(4) zen: an extract from the book "Zen Flesh, Zen Bones" (30,844 characters)

(5) lzss: a commented C program to code files using the LZSS scheme (15,072 characters)

(6) session: a transcript of text transmitted to a terminal during an edit-compile-run session (57,127 characters)

Only the file "csh" contained tabs, and in all files new lines were encoded as a single character. The first three files contained nroff formatting commands. The file "matthew" was supplied in uppercase only, and has been processed so that capitals appear only at the beginning of sentences, and in words which commonly begin with a capital letter.

matthew:

```
.PA
14Joseph got up, took the child and his mother, and left during the
night for egypt, 15where he stayed until herod died. This was done to
make what the Lord had said through the prophet come true, 'i called
my son out of egypt.'
```

short:

```
.PA
14Joseph got up, took the child and his mother, and left during the
night for egypt, 15where he stayed until herod died. This was done to
```

csh:

```
forms a pipeline.
The output of each command in a pipeline is connected to the input of the next.
Sequences of pipelines may be separated by `;', and are then executed
sequentially.
A sequence of pipelines may be executed without immediately
```

zen:

```
living a pure life.

A beautiful Japanese girl whose parents owned a food store
lived near him.    Suddenly, without any warning, her parents
discovered she was with child.
```

lzss:

```
        for (wpoint = wl; wpoint != ww; *wpoint++ = inchar())
                ;

        /* initialise the tree */
        p = nodes;
```

session:

```
                compare(posn,p^.start,ch1,ch2,dummy);
>v
   258  |
   259  |           { search tree for place to insert new node }
   260  |          repeat
```

Figure 2.1: Lines 98 to 102 of the benchmark texts

The amount of compression is measured by the *Compression Ratio* (CR), which is defined to be the size of the compressed file expressed as a percentage of the original file [Gottlieb 75]. The benchmark files were stored using the ASCII code, and the size of the original file is calculated assuming 8 bits for each character. Note that for the ASCII code one of the 8 bits is redundant, so a CR of 87.5% (7/8) can be achieved immediately by packing the characters as 7 bit codes. It will be shown in this chapter that a CR of around 70% is typical of simple TC schemes, while more sophisticated approaches can achieve CRs below 30%. To summarise the compression achieved by each scheme, the weighted average of the CRs of the benchmark texts has been calculated, using the sizes of the texts, and this value used as an *overall compression.*

The schemes which have been evaluated are described in the next sections in four groups - *ad hoc* approaches, and schemes using forms of Huffman coding, arithmetic coding, and LZ coding.

2.3 Ad hoc schemes

Ad hoc schemes can be very fast and easy to implement, but are not adaptive and usually do not achieve very good compression.

2.3.1 Macwrite

Files stored by the Macintosh word processor, Macwrite, use a simple compression scheme to save space [Young 85]. The encoding algorithm uses a string of the 15 most common characters of the language being encoded. For English, the string is " etnroaisdlhcfp". If the character to be coded is in the string, then it is coded as a 4-bit number between 0 and 14, representing its position in the string. Otherwise, it is coded as the 4-bit number 15, followed by its 8-bit ASCII code. Thus, the fifteen most common characters are coded in 4 bits, and all other characters in 12 bits. For example, the string "The tent" is coded (in hexadecimal) as "F5 4B 10 21 32". For some texts, such as text completely in upper case, this scheme will not achieve a CR less than 100%. To prevent a text being expanded, each paragraph of text is coded only if its size is reduced. An extra bit is associated with each paragraph to signal whether it has been compressed or not.

The overall compression achieved by this scheme on the benchmark files (appendix B) was 79.3%.

2.3.2 Digram Coding

A digram is a pair of characters. The principle of digram coding is to use the unallocated codes of the EBCDIC or ASCII character sets to represent common digrams. The particular method described here is Snyderman and Hunt's scheme [Snyderman 70], adapted for the ASCII character set.

The ASCII character set has 128 spare codes when stored in 8-bit bytes - usually those with the most significant bit set to one. The 128 digrams are allocated with the first character chosen from a set of 8 *master* characters and the second from a set of 16 *combining* characters. There are 128 combinations of these master and combining characters. The digrams used in the experiments were constructed using the master characters " aeiontu", and the combining characters " etaonrishdlfcmu". If the two characters in a digram had indexes in the above strings m and c respectively, then the digram was allocated the ASCII code $128 + m*16 + c$. For example, the digram coding of the string "about packing text:" is shown in Figure 2.2.

a	b	o	u	t		p	a	c	k	i	n	g		t	e	x	t	:
97	98	207		224		112	157		107	181		103	130		101	120	116	58

Figure 2.2: Digram coding example

Digram coding can never achieve a CR better than 50%, but it can never cause a file to expand. The overall compression of the digram scheme implemented on the benchmark files was 72.7%.

2.3.3 Pike's scheme

Pike's *4-bit coding scheme* [Pike 81] is similar to the Macwrite scheme, in that the first 4 bits of a code either represent a common character, or escape to some different code. Figure 2.3 shows that only thirteen common characters are represented by the first 4 bits, leaving three escape codes. Escape code '0' indicates that the next 4 bits index a common word, which is assumed to begin with a blank except at the beginning of a line. Escape code '1' indicates that the next 4 bits index one of sixteen less common characters. Escape code '2' indicates that the next 8 bits index either a rare character or a less common word.

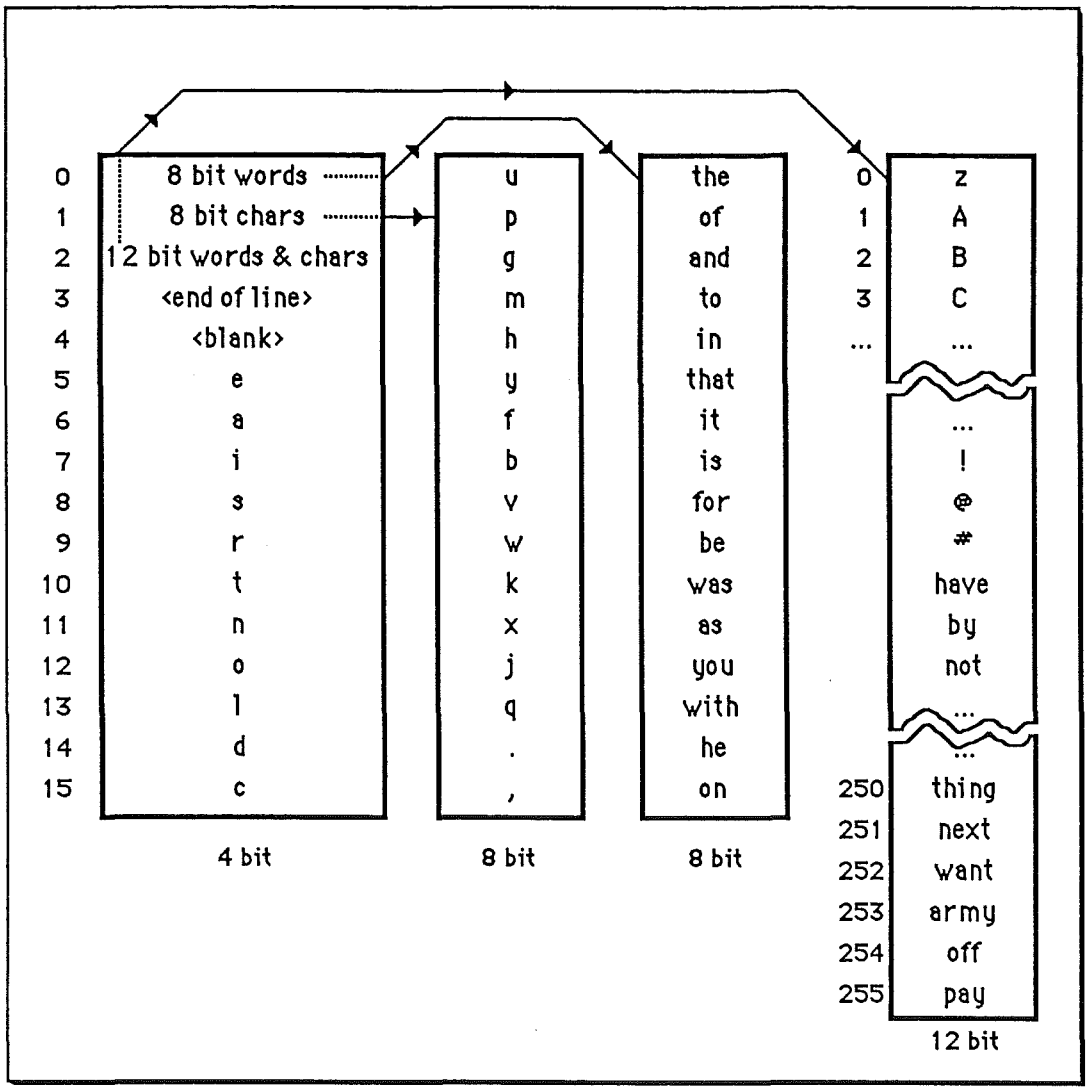


Figure 2.3: Codes for Pike's 4-bit coding scheme

Because it is advantageous to have as much text in lower case as possible, Pike includes an algorithm which removes a significant number of upper case characters from conventional texts, and from texts completely in upper case. This is done by inverting the case where upper case would be expected; in particular, when (1) the character is the first letter of the text; (2) the character is the first letter following a full stop; (3) the character is the single letter 'I'; (4) the two previous letters were upper case.

Pike's scheme achieves a CR of around 50% for English files, and performs reasonably well with program source codes, partly because many programming language

keywords are among the more common English words (e.g. of, and, to, for, if, then, while). Pike's scheme achieved an overall compression of 59.6% on the benchmark texts.

2.4 Huffman coding

The ad hoc codes so far described work on the principle that more probable events should generate shorter codes than less probable events, but no systematic method was used to determine an optimal code length for each symbol.

Shannon [Shannon 48] showed that the optimal code length for a symbol x_i with estimated probability $P(x_i)$, is $L(x_i) = -\log_2 P(x_i)$ bits. For example, for the alphabet {a,b,c,d} with probabilities P(a) = 1/2, P(b) =1/4, P(c) = 1/8, P(d) = 1/8, the optimal code lengths are L(a) = 1, L(b) = 2, L(c) = 2, L(d) = 3. There are many ways to assign codes with these lengths, but one suitable code is: C(a) = 0 , C(b) = 10 , C(c) = 110 , C(d) = 111, so for example, the string "abadcaaabb" codes to 01001111100001010.

The expected code length for an optimal code is determined using the weighted average, $\Sigma_i -P(x_i)\log_2 P(x_i)$ bits. Shannon called this value the *entropy*.

Huffman's algorithm [Huffman 52] assigns a code to each symbol with lengths as close as possible to $-\log_2 P(x_i)$, but with the necessary constraint that no code is a prefix of another code. The details are as follows. An example is given in Figure 2.4.

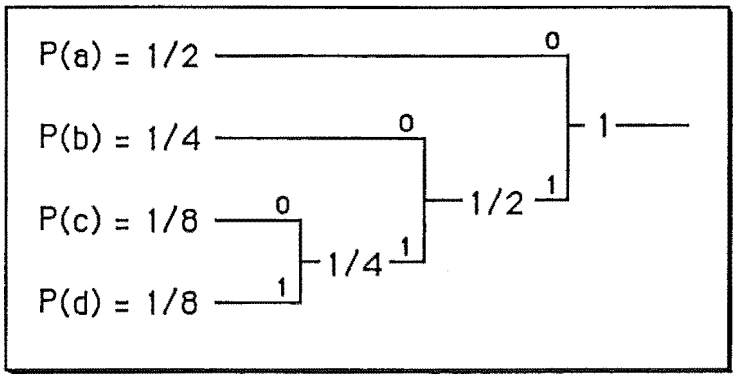


Figure 2.4: Example of Huffman's algorithm for assigning codes

The two symbols with the lowest probabilities are replaced by a composite symbol, which is given a probability equal to the sum of the probabilities of the two symbols it replaces. One of the combined symbols is labelled with a '0' and the other with a '1'. This procedure is applied recursively to the new list containing the composite symbol, until the list contains only one symbol, which has a probability of one. The code for each symbol in the original alphabet is determined by tracing the path to it from the composite symbol with the probability of one, and recording the labels allocated when symbols were combined. Thus Figure 2.4 yields the codes:

C(a)=0, C(b) = 10, C(c) = 110, C(d) = 111.

2.4.1 Implementations of Huffman coding

A straightforward way to implement Huffman coding is to make two passes over a text file, the first pass determining the probability distribution of characters and calculating the codes, and the second pass encoding the characters using the codes. Before transmitting the encoded text, the encoder must send the codes. This can be done with only two bits for each character in the input alphabet, by transmitting a preorder traversal of the Huffman tree [Horspool 86].

This two pass scheme is available as "pack" under the UNIX system, and also has been implemented on other systems with the name "squeeze" or "sq". On the benchmark files "pack" achieves an overall compression of 58.9%.

Huffman coding is quite suitable for adaptive coding. Both the encoder and decoder can maintain counts of characters seen, and estimate probabilities from these. A new code can be constructed after each character transmitted. Rather than construct the code from scratch each time, Gallager [Gallager 78] has given an algorithm which adjusts the Huffman tree as necessary as each character count is incremented by one. In practice, the codes settle down after a while, and very little adjustment is required. This scheme is available as "compact" on UNIX systems. It achieves an overall compression of 58.8% on the benchmark files.

A generalisation of Gallager's scheme was discovered apparently independently by Cormack and Horspool [Cormack 84], and Knuth [Knuth 85]. Both these papers give algorithms which allow for non-integer character counts, and decreasing character counts. This offers more flexibility in the choice of the sample used to estimate the probability distribution of characters.

2.5 Arithmetic coding

Arithmetic coding is not a particular compression scheme, but, like Huffman coding, it is a method of assigning codes to symbols based on the probability distribution of the symbols. Huffman coding is actually a special form of arithmetic coding.

Recall that the optimal length of the code for a symbol x_i with estimated probability $P(x_i)$ is $|C(x_i)| = -\log_2 P(x_i)$ bits. If the probability is a negative power of two, then $|C(x_i)|$ is an integer, and the optimum can be achieved by Huffman coding. If this is not the case, then Huffman coding can only approximate the optimum. If the probability distribution is very skew, then the average code length generated by a Huffman code can be far from the optimal average code length (entropy). This is illustrated by the coding of a two symbol alphabet in Table 2.1, where Huffman coding can only achieve one bit per symbol, while the optimum (entropy) is closer to zero as the probability distribution becomes more skew.

P(a)	P(b)	Optimal length (bits)		entropy (bits/symbol)	Av. Huffman length
		|c(a)|	|c(b)|		
$\frac{1}{2}$	$\frac{1}{2}$	1	1	1	1
$\frac{1}{4}$	$\frac{3}{4}$	2	0.42	0.81	1
$\frac{1}{16}$	$\frac{15}{16}$	4	0.093	0.34	1
$\frac{1}{256}$	$\frac{255}{256}$	8	0.0056	0.037	1
$\frac{1}{16384}$	$\frac{16383}{16384}$	14	0.000088	0.00094	1
$\frac{1}{1048576}$	$\frac{1048575}{1048576}$	20	0.0000014	0.000020	1

Table 2.1: Entropy and Huffman coding for skew distributions

Although this example is pathological, this section will discuss several models of text which produce skew distributions which are not very well coded by Huffman's algorithm.

Arithmetic coding is able to encode any probability distribution arbitrarily close to its entropy, no matter how skew the distribution. The main advantage of Huffman coding over arithmetic coding is that it is easier to understand. This may not be very significant, because software is publicly available to perform arithmetic coding and decoding [Witten 86]. All that it requires as input is a probability distribution and symbols. To evaluate a compression scheme, an implementation of arithmetic coding is not required, since the compression achieved for a given probability distribution will be (approximately) equal to the entropy. The main features of an arithmetic coder are:

(1) it will code arbitrarily close to the entropy, Σ_i-$P(x_i)\log_2 P(x_i)$,

(2) the probability distribution can change as each symbol is coded (adaptive coding),

(3) it is fast - only a few integer arithmetic operations are used to encode each symbol, and only a couple of registers are required, apart from the model which holds the probability distribution,

(4) the design of a compression scheme is divided into two distinct parts : the *model* (which is used to estimate probability distributions), and the arithmetic *coder*.

A description of arithmetic coding has been given by Witten, Neal, and Cleary [Witten 86], which includes examples, and source code for arithmetic coding in the C programming language. The following example illustrates the principle of arithmetic coding, although the reader need only know the four features of arithmetic coding given above in order to understand schemes which use arithmetic coding.

2.5.1 Arithmetic coding example

The example uses the alphabet {a,b}, with the probability distribution P(a) = 3/4 and P(b) = 1/4. The string to be coded is "aba".

During encoding two binary numbers are stored, which represent a range. As each input symbol is coded, the range will be narrowed. The narrowing range is represented graphically in Figure 2.5. At the end, any number within the range is transmitted. This description ignores the problem of the decoder recognising the end of the message.

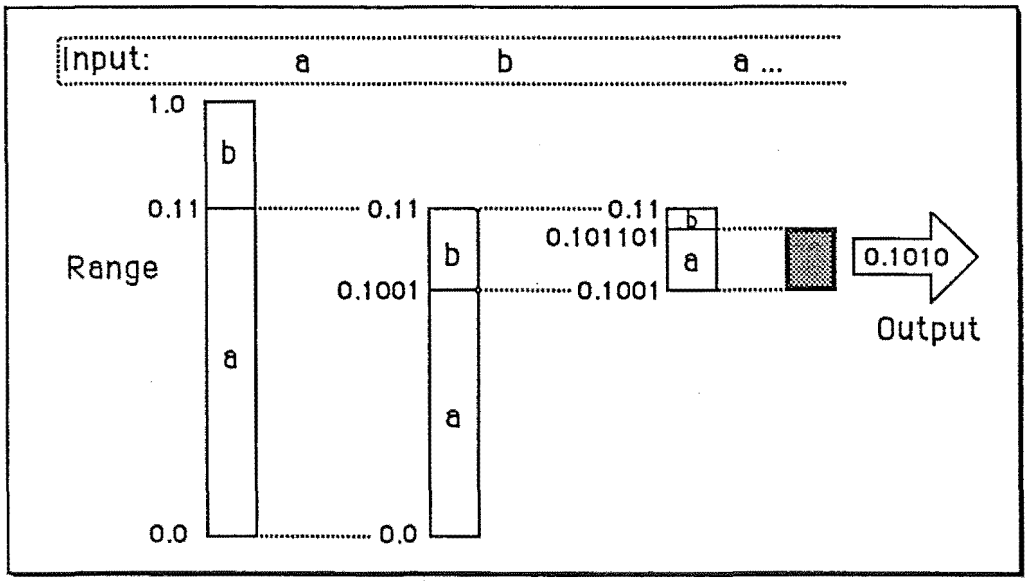


Figure 2.5: Narrowing of ranges in arithmetic coding.

Initially the range is from 0 to 1. Since every number between 0 and 1 begins with '0.', both the encoder and decoder can assume that this has already been transmitted.

The range is then divided into subranges using the probabilities of the characters as proportions i.e. (0,1) is divided into a:(0,0.11) and b:(0.11,1). The subrange of the next input character (a:(0,0.11)) is chosen for the new range.

This range is then divided up using the same proportions, giving a:(0.0,0.1001) and b:(0.1001,0.11) and the range of the next input character (b:(0.1001,0.11)) is chosen.

This time the new ranges are a:(0.1001,0.101101) and b:(0.101101,0.11) and the range a:(0.1001,0.101101) is chosen. The message is finished, and any number in the range is transmitted; 0.1010 is quite suitable.

One potential problem is that the code might get too long to work with after a while. Fortunately it is possible to transmit and discard parts as encoding proceeds. For example,

after encoding the second character above, the range is (0.1001,0.11). Regardless of further narrowing, the output must begin with 0.1, so this part can be transmitted and forgotten.

Now consider the decoder, receiving the code 0.1010. The decoder knows that the initial ranges were a:(0,0.11) and b:(0.11,1). Since the number falls in the 'a' range, the first character must be an 'a'. The new ranges are calculated to be a:(0,0.1001) and b:(0.1001,0.11). The input number is in the 'b' range, so the next character is 'b', and so on. Even if the output from the encoder is arriving intermittently, the decoder can usually deduce characters fairly shortly after they have been encoded.

In practice the arithmetic does not need to be infinitely accurate. The more accurate it is, the closer to optimal the codes are. If the ranges are stored as 16 or 32 bit integers, coding will be very close to optimal (typically within 0.5% for 16 bits [Witten 85]). So the final algorithm requires only a few arithmetic operations (multiply, add, shift) for each input symbol, and the only memory used is two registers! Also, the probabilities may change adaptively at each step.

If the probabilities supplied to an arithmetic coder are negative powers of two, then the codes generated are the same as those generated by a Huffman coder.

2.5.2 Models for arithmetic coding

Since arithmetic coding can provide coding as close to optimal as desired, the real challenge now is to find feasible models of text to exploit this. A simple model might calculate the proportion of occurrence of each symbol in the text, and use this proportion to estimate the probability of the symbol. This is the model usually used for Huffman coding schemes such as "pack". In Table 2.2, the scheme "zero-order" is an implementation of this model to estimate the compression it would achieve with arithmetic coding. The results show that only a slight improvement is achieved over Huffman coding (an overall compression of 58.1%, compared with 58.9%).

A more sophisticated model can be constructed by observing that the probability of a symbol in a text is strongly influenced by the symbols immediately preceding it. For example, if the letter 'q' has just been encountered, the probability of a 'u' occurring next is very high, and the probability of an 'x' is very low. This approach of predicting a symbol from its immediately preceding symbols is called *Markov modelling*. A model which uses one symbol of prior context is a *first-order Markov model*; a model with two symbols of prior context is a *second-order Markov model*, and so on. It follows that a zero-order Markov model uses no prior context for prediction. In Table 2.2, the overall CR of a first-order Markov model is 41.0%, compared with 58.1% for a zero-order Markov model. It would appear that the larger the context used, the better. Unfortunately, the use of very large contexts is difficult in practice because a large sample would be required to accurately estimate suitable probabilities, and a large amount of memory would be required to store them. One solution is *variable-order Markov modelling*, where the model switches between different size contexts using the largest context for which probabilities can be safely estimated [Roberts 82].

Four models suitable for arithmetic coding are now described. The first two use a variable-order Markov model (VOMM).

2.5.3 Double-adaptive file compression (DAFC)

This adaptive scheme [Langdon 83b] is a compromise between zero- and first-order Markov modelling. Instead of recording the counts of characters in *every* first-order context, only 31 common first-order contexts are chosen, and counts recorded for these. Counts for other contexts are recorded in one *lump* zero-order context. The 31 most common contexts are chosen as the first 31 symbols in a text to be encountered more than some minimum number of times (typically 50). In addition to these contexts, a special *run mode* is entered when the two previous symbols are the same. The model remains in this mode as long as the same symbol is encountered, encoding each symbol with a single bit,

'1'. When a different symbol is encountered, a '0' is transmitted, and the model returns to the normal *symbol mode*. The *run mode* is actually a type of second-order Markov context.

The compression achieved by DAFC is usually in between that of the zero- and first-order models. The overall compression on the benchmark files was 46.4%, compared with the zero- and first-order CRs of 58.1% and 41.0% respectively.

2.5.4 Prediction by Partial Matching (PPM)

This adaptive scheme [Cleary 84b] uses a variable-order Markov model so that it can use large contexts where possible, but is also capable of shifting to smaller contexts, particularly in the early stages of encoding when the model has not built up much information. To encode each character, the longest context for that character which has occurred before in the text is used, up to some maximum, typically of four characters. For example, if the next character is 'n' and the previous four characters are "ssio", the PPM scheme calculates the proportion of times 'n' has occurred in the context "ssio" in the previously seen text, and uses this as a probability to encode 'n'.

There are two special cases. The first is when "ssio" has never occurred before. Fortunately both the encoder and decoder will be aware of this, and automatically shift to the third-order context, "sio". If this context has not occurred before, the size of the context is reduced until it has. At the very worst, the zero-order context is reached, which has always been seen before.

The second special case is when the context has been seen before, but the character has never been seen in that context. The decoder will not be aware of this, so in this case the encoder transmits a special escape message which tells the decoder to shorten the context by one character. A small amount of *code space* (probability) is set aside so that this escape message can be transmitted. As with the first special case, this process is repeated until the character has been seen before in the context. If the character has not been seen before in

the zero-order context, then another escape message is sent, and the character is transmitted explicitly.

For efficient implementation, the model is stored in a trie data structure, allowing fast searching for contexts. Considerable amounts of memory are required, particularly when a large maximum context is used. For the benchmark files, the optimum maximum context was 4 or 5 characters, except for "session", which was compressed best with a maximum context of 7 characters. The larger context for "session" was due to many repeated phrases in the text. The PPM scheme gave better compression than any other scheme evaluated, for every benchmark text except "short" with the first-order and MTF schemes. The overall compression achieved by PPM was 28.5%.

2.5.5 Dynamic Markov Compression (DMC)

In this adaptive model [Cormack 87], the input is usually treated as a stream of bits, rather than characters. The model is a Finite State Machine, where each state has two output transitions (on 0 and 1). Each transition out of a state has a probability estimated from the number of times it is used compared with the number of times the state has been visited.

For each bit in the input, the encoder uses arithmetic coding of these probabilities to tell the decoder whether to take the 0 or 1 transition from the current state. Both the encoder and decoder then update the counts of the transition, and move to the next state.

The model is made adaptive by *cloning* new states when one transition is found to be heavily used. For example, if part of the model is as shown in Figure 2.6a, and the transition from state u on bit e to state t is heavily used, then a new state, t' is created, and the new model is as shown in Figure 2.6b.

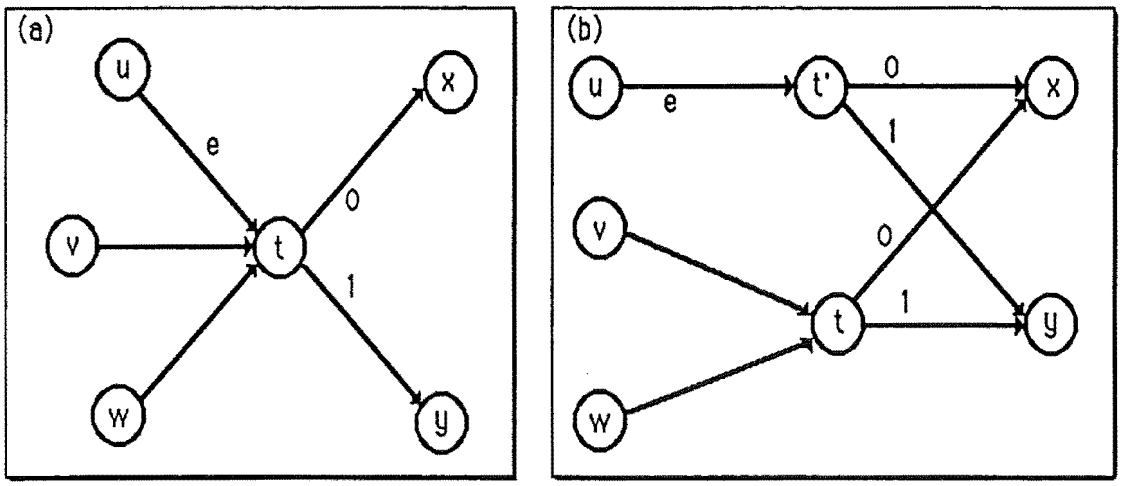


Figure 2.6: The DMC cloning operation

The model can begin with just one state (Figure 2.7a) which rapidly grows to form a good model. More sophisticated initial models can be used to achieve small improvements in the CR. For the initial zero-order Markov model in Figure 2.7a, the first cloning operation always results in the first-order Markov model of Figure 2.7b.

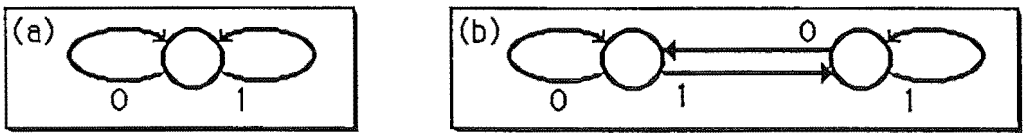


Figure 2.7: A simple initial model for DMC, and the result of the first cloning operation

This scheme is described in more detail in chapter 4. The compression achieved by DMC on the benchmark texts was bettered only by the PPM scheme, except for the file "short". DMC achieved an overall compression of 30.6%.

2.5.6 Move To Front (MTF)

MTF is an implementation of *A locally adaptive data compression scheme* [Bentley 86], with the design choices made to achieve as much compression as possible. It is the result of work performed by Dr. A.M.Moffat. In this scheme, the text is treated as a series of words separated by blanks and punctuation. Recently transmitted words are stored in a list (cache). If the word to be transmitted is already in the list, only its position in the list is

transmitted, and it is moved to the front of the list. The nearer the position is to the front of the list, the shorter its code is. If the word is not in the list, it is transmitted explicitly, and added to the front of the list. A separate list can be maintained in the same way for blanks and punctuation. The lists typically have a maximum size of a few hundred words, so words are removed from the back of the list when it is full.

As an example, when transmitting:

"the car on the left hit the car I left",

after the first 3 words have been encoded, the list is

<front> on car the <back>

and the word "the" is encoded as position 3 in the list. The entire message is encoded as:

the car on 3 left hit 3 5 I 5

This example illustrates the principle of the MTF scheme: if a word has been recently used then it will be near the front of the list and therefore have a short encoding. This algorithm has aspects in common with paging algorithms for virtual memory systems; both address the problem of choosing what should be retained in memory by predicting what is most likely to be reused in the near future.

The task of maintaining such a list is handled by an efficient algorithm supplied by Bentley *et. al.* There are other design issues for which several solutions are offered. These are:

Lexical analysis: In order to break a text up into words, a useful definition for a *word* is needed. A simple approach is to treat each character as a word, or to use contiguous alphabetic characters. More sophisticated definitions might be helpful with different types of text. The MTF implementation breaks the text into two types of word: *English words*,

which are longest sequences of alphabetic characters, and *spaces*, which are longest sequences of non-alphabetic characters.

List organisation discipline: The method used in the MTF implementation is to move a word directly to the front of the list when it is used. Another approach would be to move the word one place nearer the front.

List length: In the MTF implementation, the maximum list length was set to be so large that it was never filled. Bently *et. al.* found that a maximum length of 512 words performed well, although no experiments were performed with larger lists.

Encoding list position: Bentley *et. al.* used Huffman coding to encode the list position, with probabilities being estimated in a first pass. It turns out that arithmetic coding achieves much better compression, and this was used in the MTF implementation, with probabilities being estimated adaptively by counting references to each position in the list.

Transmission of new words: Bentley *et. al.* signal a new word by transmitting the list position one larger than the number of words in the list, and then sending the characters explicitly. The MTF implementation prefixes each word with a flag which indicates whether the word is in the list, or if it is new. The flag probabilities are estimated adaptively, the length of the new word is transmitted (using adaptive probabilities for the size), and the explicit characters are transmitted using an adaptive zero-order code.

The MTF scheme achieved an overall compression of 33.3% on the benchmark texts.

2.6 Ziv-Lempel (LZ) coding

Some confusion exists about what *LZ coding* is. Several compression schemes have been labelled as *LZ coding*, but each is subtly different from the others. A definition which includes all *LZ schemes* is "a compression scheme in which groups of characters (phrases) are replaced with a pointer to a previous occurrence of the phrase in the text". Storer and Szymanski [Storer 82] have labelled this class of compression schemes *Original Pointer*

Macro restricted to Left Pointers (OPM/L). An OPM/L scheme replaces a substring in a text with a *pointer* to a previous (*left*) occurrence of the substring in the (*original*) text.

For example, if pointers of the form (*m,l*) represent the phrase of *l* characters starting at position *m*, the string

```
1  2  3  4  5  6  7  8  9 10 11 12
a  b  b  a  a  b  b  b  a  b  a  b  could be coded as

a b b a (1,3) (3,2) (8,3),    and

1  2  3  4  5  6  7  8
a  b  a  a  a  a  b  a     could be coded recursively as

a b a (3,3) (2,2)
```

Restricting pointers to the left makes decoding straightforward. Two factors which distinguish between versions of LZ coding are whether there is a limit to how far back a pointer can reach (window), and which substrings within the window may be pointed to.

Figure 2.8 shows a "family tree" for the main LZ schemes. Each scheme has been assigned a distinguishing label. An asterisk (*) indicates that the scheme was not actually designed for compression but for evaluating the "complexity" of a string (see section 2.6.1). The pair of letters underneath describe the choice of what can be referenced by pointers, as follows:

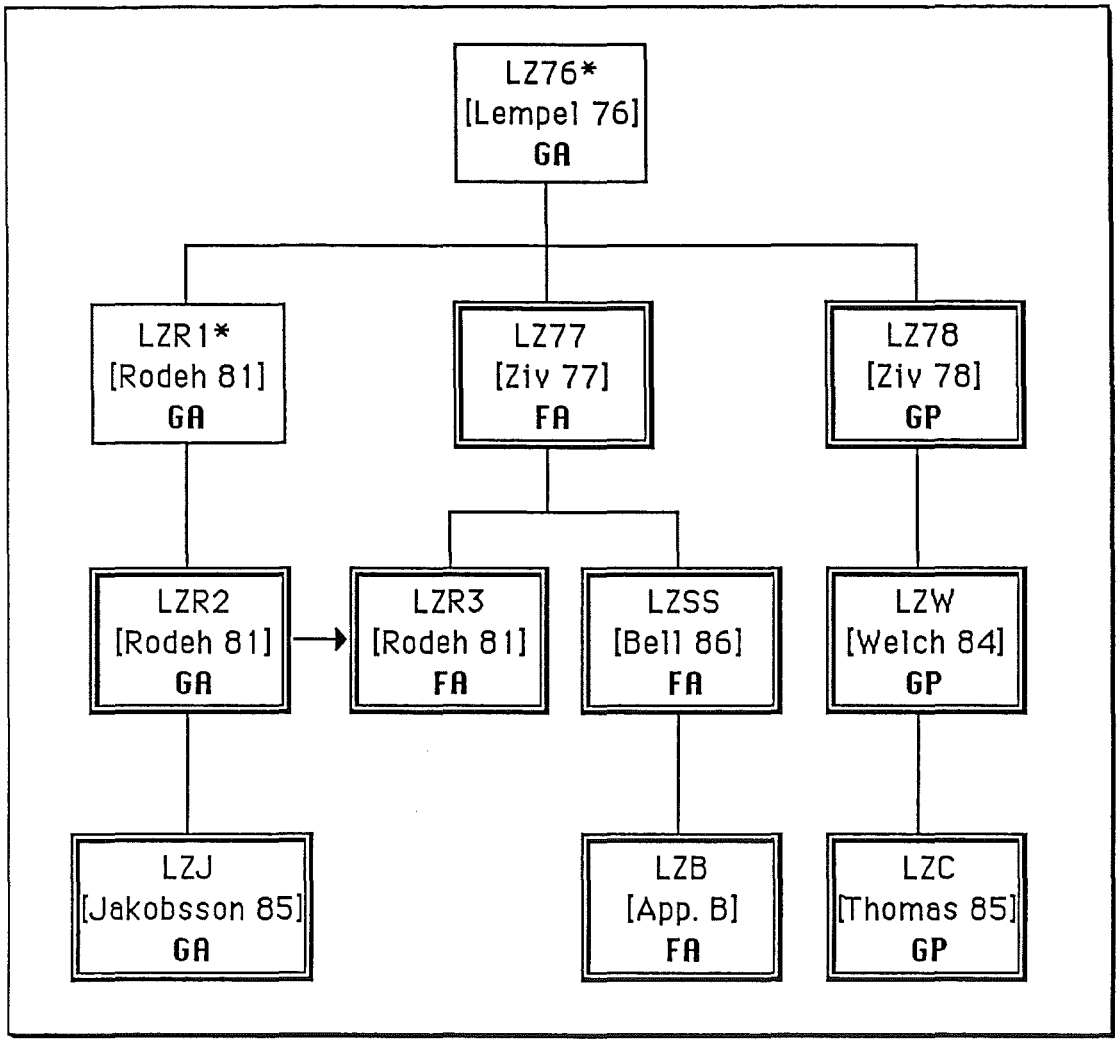


Figure 2.8: A family tree of LZ schemes

Reach of pointers (windows):

G: pointers may index substrings anywhere in the text seen so far i.e. a *growing* window. Occasionally memory constraints or an increasing CR may cause all the text seen so far to be discarded, and coding continues as if for a new text.

F: pointers may index substrings in only a *fixed* size window of immediately preceding characters.

Target of pointers:

P: the text in the window is broken up into *phrases*, and a pointer selects one of these phrases.

A: pointers can reference *any* substring in the window, so the pointer has two components: the position of the substring, and the length of the substring.

Each combination of these choices usually represents some compromise between speed, memory requirements, and compression. The growing window offers better compression by making more substrings available. However, as the window grows larger, the encoding speed becomes worse because of the time taken to search for matching substrings; compression may get worse because pointers must be larger; and if memory runs out, the window must be discarded, giving poor compression until the window grows again. The fixed size window avoids all these problems, but has less substrings available to point to.

Breaking the window into phrases allows pointers to be smaller, and searching with a fast trie or hashing algorithm is straight-forward. However, considerably fewer substrings are available this way than when any substring can be referenced.

The two main sub-families of LZ schemes are those developed from LZ77 and LZ78. The LZ77 family use a fixed window and allow any string to be referenced by a pointer, while the LZ78 family use a growing window, and break it into phrases. In general, the LZ77 family gives better compression, while the LZ78 family gives faster encoding. Each LZ coding scheme is now described in more detail.

2.6.1 LZ76

LZ76 [Lempel 76] is not a compression scheme, but an algorithm to evaluate the *complexity* (or *randomness*) of a string (text). This is done by breaking a text up into substrings, or phrases, which is the principle behind LZ coding.

LZ76 is described by Lempel and Ziv as a "simple self-delimiting learning machine which, as it scans a given n-digit (n-character) sequence $S = s_1 s_2 \ldots s_n$ from left to right, adds a new word to its memory every time it discovers a substring of consecutive digits not previously encountered. The size of the compiled vocabulary, and the rate at which new words are encountered along S, serve as the basic ingredients in the proposed evaluation of the complexity of S".

Of course, the shortest substring of characters not previously encountered is the longest sequence already encountered plus one character. For example, the string:

S = a a a b b a b a a b a a a b a b , is parsed as

LZ76(S) = a . aab . ba . baa . baaa . bab

Shortly after the LZ76 paper was written, Rodeh Pratt and Even [Rodeh 81] wrote a paper which contained three distinct algorithms, the last two being compression schemes. The three algorithms have been labelled LZR1, LZR2 and LZR3.

2.6.2 LZR1

Lempel and Ziv were more concerned with the theoretical properties of LZ76, than with an efficient algorithm to compute it. A straightforward implementation would take $0(n^2)$ time, because the search for a previous occurrence of a substring must inspect every character of the text seen so far. LZR1 [Rodeh 81] is an implementation of LZ76 which uses McCreight's data structure [McCreight 76] to process text in 0(n) time. McCreight's data structure is derived from a trie data structure.

2.6.3 LZR2

LZR2 is the LZR1 algorithm modified to perform compression. Each phrase which is parsed by LZR1 (i.e. LZ76) is replaced in the output by the triple $<i,j,a>$, where the length of the phrase is $j+1$, a is the last character of the phrase, and i and j give the position and length of the previous occurrence of the phrase (less the character a). The position (i) is

given by numbering the first character of the text one, the second character two, and so on. For the string used as an example for LZ76,

S = a a a b b a b a a b a a a b a b,

LZ76(S) = a . aab . ba . baa . baaa . bab, and

LZR2(S) =

<0,0,'a'><1,2,'b'><4,1,'a'><5,2,'a'><7,3,'a'><5,2,'b'>

Because the values of i and j can grow arbitrarily large, a variable length coding of the integers is used to represent them. The method used is from [Even 78], and has been labelled RPE here. It is similar to Elias' code C_ω [Elias 75]. Both are described in Appendix C.

A problem with LZR2 is that it requires more and more memory as encoding proceeds. LZ77 solves this problem by using a fixed-size window, and Rodeh, Pratt and Even used this idea to produce LZR3. LZ77 will be described first.

2.6.4 LZ77

An LZ77 encoder [Ziv 77] is parameterised by N, the size of the window on the text, and F, the maximum length of a substring that may be replaced by a pointer. At each step of the encoding a section of the input text is available in the window of N characters. Of these, the first N-F characters have already been encoded and the last F characters are the *lookahead buffer*.

For example, if the string "abcabcbacbababcabc···" is being encoded with the parameters N = 11 and F = 4, and character number 12 is to be encoded next, the window is as shown in Figure 2.9.

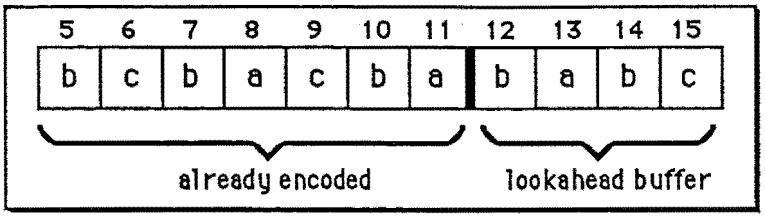


Figure 2.9: LZ77 encoding

Initially the first N-F characters of the window are (arbitrarily) blanks, and the first F characters of the text are loaded into the lookahead buffer.

The already encoded part of the window is searched to find the longest match for the lookahead buffer. The match may overlap with the lookahead buffer, but obviously cannot be the lookahead buffer itself. In Figure 2.9, the longest match for the "babc" is "bab", which starts at character 10.

The longest match is then coded into a triple <*i*,*j*,*a*>, where *i* is the offset of the longest match from the lookahead buffer, *j* is the length of the match, and *a* is the first character which did not match the substring in the window. In the example, the output triple would be <2,3,'c'>. The window is then shifted right *j*+1 characters, ready for another coding step.

A window of moderate size, typically $N \leq 8192$, can work well for a variety of texts because:

(1) Common words and fragments of words occur regularly enough in a text to appear more than once in a window. Examples are: in English "the", "of", "pre-", "-ing"; in a source program, keywords "while", "if", "then".

(2) Specialist words tend to occur in clusters, for example, words in a paragraph on a technical topic, or local identifiers in a procedure of a source program.

(3) Less common words may be made up of fragments of common words

(4) Runs of characters are coded compactly. For example, k blanks may be coded recursively as <?,?,' '> <1,k-1,?>.

The amount of memory required for encoding and decoding is limited to the size of the window. The offset (*i*) in a triple can be represented in $\lceil \log_2(N-F) \rceil$ bits, and the number of characters (*j*) covered by the triple in $\lceil \log_2 F \rceil$ bits. The time taken at each step is bounded to N-F substring comparisons, which is constant, so the time used for encoding is O(*n*) for a text of size *n*.

Decoding is very simple and fast. The decoder maintains a window in the same way as the encoder but, instead of searching for a match in the window, it copies the match from the window using the triple given by the encoder.

The main disadvantage of LZ77 is that, although the encoding step requires O(1) time, a straightforward implementation can require up to (N-F)*F character comparisons, typically in the order of several thousands. LZ77 is therefore best for the situation where a file is to be encoded once (preferably on a fast computer) and decoded many times, possibly on a small machine. Examples of these situations are on-line help files, manuals and news, de-centralised data bases [Urrows 84], teletext [Money 79], and electronic books [Weyer 85].

LZ77 achieved an overall compression of 45.3% on the benchmark texts.

2.6.5 LZR3

LZR3 was developed from LZR2, using Ziv and Lempel's finite window (LZ77). The data structure used in LZR2 to represent the window does not easily allow deletions as characters leave the window, so LZR3 uses several copies of the data structure to index the window. Each index covers blocks of N characters, and these blocks overlap so that the characters in any window of size N are in no more than three indexes. Figure 2.10 shows a window covered by indexes number 2, 3 and 4. Characters are added to the relevant indexes when they enter the window. An index is discarded when none of its characters are

in the window, so at most three indexes must be stored at once. To find a longest match in the window, the first two indexes must be searched (the overlapping ensures that the third index is only useful after the first index is discarded). A probe into the first index must be careful to ignore strings no longer in the window. The better of the matches produced by the two indexes is chosen.

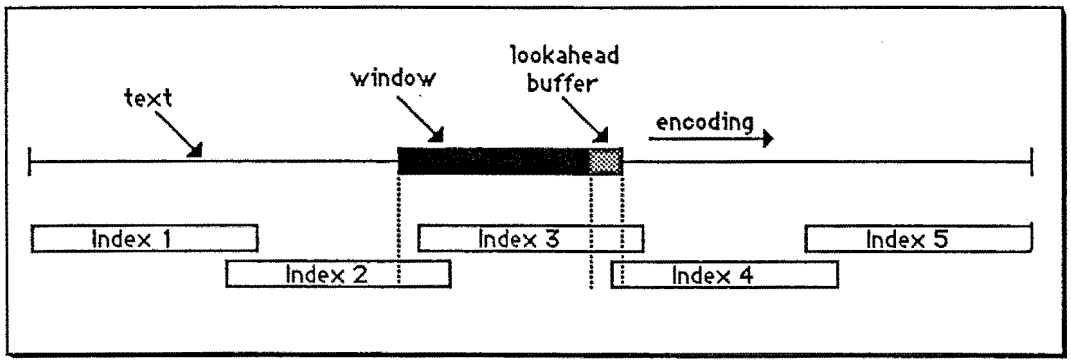


Figure 2.10: LZR3 coding

LZR3 is not of practical use because a straightforward implementation of LZ77 is O(n) in time and memory usage, and Rodeh, Pratt and Even were motivated by LZ76, which was $O(n^2)$ in time, or $O(n^2)$ in memory if a trie data structure was used. The LZ77 algorithm is easily implemented with a trie structure, which allows deletion of any character leaving the window. A trie is probably just as fast, and much simpler to implement than LZR3. A binary tree can also be used to implement fast searching, and allows character by character deletion. This is given in chapter 7.

LZR3 may still be useful as a basis for faster encoders for LZ77, if the index used does not easily facilitate deletion (e.g. hashing).

2.6.6 LZSS

The output of the LZ77 scheme is a series of triples, which can also be viewed as a series of strictly alternating pointers and characters. In what follows, the triple $<i,j,a>$ will now be denoted by the pointer (i,j), and the character a. In Storer and Szymanski's work on OPM/L schemes [Storer 82], they suggest that the Ziv-Lempel algorithms use a free mixture of pointers and characters; a character being used only when a pointer would take

more space than the characters it codes. In [Bell 86] the LZ77 scheme has Storer and Szymanski's suggestion incorporated to produce the LZSS algorithm. Suppose a pointer uses the space of p unencoded characters. The LZSS algorithm is:

```
while lookahead buffer not empty do
     get a pointer (offset,length) to the longest match
          in the window for the lookahead buffer
     if length > p then
          output the pointer (offset,length)
          shift window length characters
     else
          output first character in lookahead buffer
          shift window one character
```

An extra bit is added to each pointer or character to distinguish between them. The output is packed so that there are no unused bits.

Implementations of the LZSS (and LZ77) encoders and decoders can be simplified by numbering the input text characters modulo N. The window is an array of N characters. To shift in character number r (modulo N), it is simply necessary to overwrite element r of the array, which implicitly shifts out character r-N (modulo N). Instead of an offset, the first element of an (i,j) pointer can be a position in the array (0 .. N-1). This means that i is capable of indexing substrings which start in the lookahead buffer. These F unused values of i cause negligible deterioration in the compression ratio provided that F << N, and they can be used for special messages such as *end of file.*

The first element of a pointer can be coded in $\lceil \log_2 N \rceil$ bits. Because the second element of the pointer can never have any of the values $0,1, \cdots ,p$, it can be coded in $\lceil \log_2 (F - p) \rceil$ bits. Including the pointer flag bit, a pointer requires

$$1 + \lceil \log_2 N \rceil + \lceil \log_2 (F - p) \rceil \text{ bits.}$$

Input characters are assumed to be ASCII, although the scheme is easily adapted for EBCDIC and other codes. An output character requires one flag bit plus 7 bits to represent an ASCII character, a total of 8 bits.

Further details and experiments with LZSS are reported in [Bell 86] (see appendix A). A fast encoding algorithm is given in chapter 7. Experiments show that using N = 8192 and F = 16 typically yields good compression for a range of texts, and that the choice of N and F is not too critical. The LZSS scheme with N=8192 gave an overall compression of 39.2% on the benchmark texts compared with 45.3% for LZ77.

An improved version of LZSS, developed by the author and called LZB, will be given in chapter 6. LZB achieved an overall compression of 37.7%.

2.6.7 LZ78

Because encoding for LZ77 was very slow, Ziv and Lempel developed LZ78 [Ziv 78], which allows fast encoding, and still offers compression comparable with LZ77.

Instead of allowing pointers to reference any string which has appeared previously, the text seen so far is parsed into phrases, where each phrase is the longest *phrase* seen previously plus one character. Each phrase is encoded as an index to a previous phrase, plus the extra character. The new phrase is then added to the list of phrases that may be indexed.

For example, the string S = "aaabbabaabaaabab" is divided into the 7 phrases:

```
   a  a a  b  b a  b a a  b a a a  b a b
0  1    2  3    4      5         6       7
```

and coded as:

LZ78(S) = (0)a (1)a (0)b (3)a (4)a (5)a (4)b

LZ78 uses a growing window, so more and more phrases are stored as encoding proceeds. To allow for an arbitrarily large number of phrases, the size of a pointer must grow as more phrases are parsed. When ρ phrases have been parsed, a pointer is allocated $\lceil \log_2 \rho \rceil$ bits. Searching is implemented efficiently by inserting each phrase in a trie data structure [Langdon 83a]. The process of inserting a new phrase in the trie will yield the

longest phrase previously seen. LZ78 achieved an overall CR of 49.2% on the benchmark texts, compared with 45.3% for LZ77.

2.6.8 LZW

Welch's scheme, LZW [Welch 84], improves on LZ78 in several ways. The output of LZW contains pointers only. This is achieved by initialising the list of phrases to every character in the input alphabet. The last character of each new phrase is encoded as the first character of the next phrase. Some difficulties arise if that character is required to encode the next phrase, but these can be dealt with.

Transmission of pointers is simplified, and sped up, by using a constant size of (typically) 12 bits. After 4096 phrases have been parsed, no more are added to the list. LZW achieved an overall CR of 47.4% on the benchmark files compared with 49.2% for LZ78.

2.6.9 LZC

LZC [Thomas 85] is the program "compress", available on UNIX systems. It began as an implementation of LZW, and has been modified several times to achieve better compression, and to run faster. The result is a high-performance scheme which is considered the most practical currently available.

An early modification was to go back to the variable length pointers of LZ78. The part of the program which stores pointers was coded in assembler language to maintain a good speed. The parameter BITS gave a maximum length for pointers (typically BITS=16, but less for small machines) to prevent the phrase list overflowing memory. A further modification monitors the CR once the phrase list is full. If the CR is deteriorating, the phrase list is cleared, and rebuilt from scratch.

A hashing function is used to search the phrase list, so the implementation is very fast. LZC achieves an overall CR of 40.7%, comparing very favourably with the other LZ schemes.

2.6.10 LZJ

An unusual form of LZ coding was published recently by Jakobsson [Jakobsson 85], and has been labelled LZJ. The LZJ encoder and decoder use a trie data structure to adaptively store substrings from the already encoded part of the text. The depth of the trie is limited to h characters (typically h=6), and it may not contain more than H nodes (typically H=8192). Each node in the trie has a unique number from 0 to H-1. Initially the trie contains each character in the alphabet. At each encoding step the trie is searched using the characters about to be encoded. When the path down the trie is blocked, the number of the last node encountered is transmitted (using $\log_2 H$ bits), from which the decoder can deduce the path down the trie. For each character encoded, the substring of length h which is terminated by that character is then inserted in the trie.

This encoding step is repeated until the trie contains H nodes. At this point the trie is full, and the encoder could choose not to insert any more strings in the trie. Slightly better compression is achieved if the trie is "pruned" when it is full, by removing all nodes which have been visited only once. Encoding proceeds, and each time the trie becomes full, it is pruned again.

LZJ encoding has the advantages of a fixed size output code, and a fast trie data structure for encoding. However, the pruning algorithm slows down the coding considerably, and large amounts of memory are required for good compression. Experiments have shown that the best performance usually occurs when H is the same order of magnitude as the number of characters in the text being compressed. An overall CR of 42.9% was achieved using H=8192. This can be improved by a few percent by using H=131072, but the time and memory requirements are very large.

2.7 Summary

Table 2.2 gives the Compression Ratios for each of the schemes described, measured for the benchmark texts of section 2.2. The schemes are ranked by their overall compression. The results show that three schemes which use arithmetic coding (PPM, DMC, and MTF) usually gave significantly better compression than any other scheme, with PPM achieving the best overall compression. To the knowledge of the author, none of these three arithmetic schemes has yet been implemented to perform compression and decompression - the CRs were evaluated from the probability distributions generated by the schemes.

All of the LZ schemes, with the possible exception of LZJ, are fully implemented, and so are the best available *working* compression schemes. Of these, LZB gave the best compression, and also has the advantage that decoding is fast and requires little memory. LZC achieved almost as much compression as LZB, and offers significantly faster encoding.

This seemingly diverse range of schemes will now be put into the unifying framework of *variable-order Markov modelling*. In the process of doing this, the means by which each scheme achieves compression will be made clearer, and the empirical ranking will be verified.

		section	matthew	short	csh	zen	lzss	session	overall
length (characters)			139521	4510	60997	30844	15072	57127	308071
Macwrite		*2.3.1*	73.8	72.3	74.6	70.5	69.2	105.8	79.3
digram		*2.3.2*	69.9	69.6	70.7	67.9	65.4	86.2	72.7
Pike		*2.3.3*	52.8	53.1	58.0	50.9	64.1	82.0	59.6
pack		*2.4.1*	59.5	59.6	58.6	56.5	44.2	62.6	58.9
compact		*2.4.1*	58.7	59.5	59.7	56.7	44.3	62.7	58.8
zero-order		*2.5.2*	58.2	57.2	59.0	55.8	43.4	62.1	58.1
LZ78		*2.6.7*	50.4	65.5	52.4	57.6	38.6	39.6	49.2
LZW		*2.6.8*	47.9	64.7	49.7	50.6	33.8	44.3	47.4
DAFC		*2.5.3*	45.9	59.5	48.0	48.4	37.0	46.5	46.4
LZ77	N=8192	*2.6.4*	49.7	65.4	47.5	56.5	30.7	28.4	45.3
LZSS	N=2048	*2.6.6*	49.2	52.8	46.5	54.5	28.1	27.3	44.2
LZJ	H=8192	*2.6.10*	44.7	57.2	43.3	49.0	32.7	36.4	42.9
first-order		*2.5.2*	42.6	40.5	43.4	43.8	27.4	36.6	41.0
LZC	BITS=16	*2.6.9*	41.5	56.9	42.9	48.9	31.5	32.9	40.7
LZSS	N=8192	*2.6.6*	43.3	54.7	40.8	48.3	25.6	25.0	39.2
LZB	N=8192	App. B	41.9	49.1	39.5	47.1	23.3	23.5	37.7
MTF		*2.5.6*	34.2	43.9	35.1	39.6	24.2	27.2	33.3
DMC		*2.5.5*	31.9	47.8	32.5	39.4	23.2	21.3	30.6
PPM		*2.5.4*	29.3	45.3	30.1	37.7	21.9	20.3	28.5

Table 2.2: Compression Ratios (%) for TC schemes described in chapter 2.
Where a parameter is not given, the optimal value has been used.

Chapter 3

Variable-order Markov models and Finite Context Automata

3.1 Variable-order Markov models

It has already been shown that arithmetic coding can encode a text optimally (almost) for any probabalistic model of the text. In order to achieve good compression the model must predict each symbol as accurately as possible. One class of model which seems to do this well is that of variable-order Markov models, which are defined as follows.

3.1.1 Definition

A *variable-order Markov model* (VOMM) is any model where each symbol is predicted by a finite number of immediately preceding symbols (Markov context), with the size of the context chosen to be as large as possible. Escape messages may be introduced to ensure that the size of the context used by the encoder and decoder are the same. Both adaptive and non-adaptive models are included.

3.1.2 Comments

The class of VOMMs is epitomised by the powerful PPM scheme, but by imposing suitable restrictions, it includes schemes as weak as ASCII coding. The main restriction on the size of the context be will caused by the sampling error introduced by using a context for which the model has too few samples. The lack of samples might be due to (1) not having seen a context before (inexperience), (2) discarding samples due to memory restrictions (forgetfulness), or (3) discarding or ignoring samples to save time (laziness).

A VOMM must start with some initial model, which may change as the text is coded. The initial model may be based on simplistic assumptions, such as "every character is equiprobable"; it may be based on other texts already seen; or it may be generated by an initial pass through the text about to be encoded.

The class of VOMMs includes models where the size of the context used for prediction is constant, that is, fixed-order Markov models.

The optimal method of encoding predictions made by a VOMM is arithmetic coding, but other encodings may be used. In particular, Huffman coding is included, which is a subset of arithmetic coding.

Despite the good compression achieved by VOMMs such as PPM, this type of model can only approximate the true behaviour of English text because it does not attempt to identify any grammatical structure in a text. For example, the fact: "opening quotations (and parentheses) are almost always followed by closing quotations (and parentheses)" cannot be captured in a Markov model. Also, a verb may have a lot of influence on a noun, regardless of the number of adjectives in between. For example, in "I stroked the fat cat" and "I stroked an old black cat", a Markov model would give the greatest weighting to the adjectives "fat" and "black" to predict the word "cat", when it is really predicted by the verb "stroked".

The main result presented in this thesis is that none of the compression schemes described in chapter 2 is more powerful than a VOMM, indicating that more powerful models should be investigated to achieve better compression.

In Ozeki's work on coding linguistic information, he agrees that "... a Markovian source is too poor a model viewed from the knowledge of current linguistics" [Ozeki 74b], and goes on to investigate the use of a Context Free Grammar as a model [Ozeki 74a, 74b, 75]. An interesting result of this work is that the sentence generation process for a CFG is a Markov process, indicating that a well developed understanding of Markov models will still be necessary in the future when more sophisticated models are used.

The proofs that TC schemes use variable-order Markov modelling are generally constructive, and are intended to offer insight into Markov modelling in its many forms.

In order to unify the TC schemes described in chapter 2 as forms of variable-order Markov modelling, several types of proof are used. Some of the schemes immediately fit the definition of a VOMM, so no proof is required. These are "pack", "compact", DAFC and PPM.

MTF uses four zero-order Markov models for coding, switching between them using flags (escape codes). Each *word* is coded using a zero-order Markov context, since it is assigned a probability according to its list position. A second zero-order Markov model is used for *spaces*. The flag which signals that a word (or space) is not in the list switches to another zero-order Markov model for the explicit spelling of the word or space, with separate models being used for each.

To prove that DMC uses a VOMM, a subset of Finite State Automata, called Finite Context Automata, is now introduced, and is shown to be a form of variable-order Markov modelling. A proof that DMC generates only FCAs is given in chapter 4. The remaining TC schemes will be shown to belong to the class of VOMMs in chapter 5.

3.2 Finite Context Automata

Finite State Automata

A Finite State Automaton (FSA) is a quadruple (S,A,μ,s), where S is a finite set of states, A is a finite set of symbols (alphabet),

$$\mu : S \times A \rightarrow S$$

is the next state function, and $s \in S$ is the starting state.

Let A* be the set of all strings, including the empty string, with elements drawn from A. The empty string is denoted as Λ. For convenience, μ is extended to A*:

$$\mu : S \times A^* \rightarrow S$$

by the recursive definition:

$$\mu(s,\Lambda) = s,$$
$$\mu(s,p.a) = \mu(\mu(s,p),a) \quad \text{for } p \in A^*, a \in A.$$

The reader is referred to [Kain 72] for more information about FSAs.

Finite Context Automata

Let F = (S,A,μ,s) be a deterministic FSA. For a machine F there will be a set (possibly empty) of strings $1 \in A^*$ that have the special property of forcing F into a particular state, no matter what original state F is in i.e. $\mu(s_i,1) = \mu(s_j,1)$ for $s_i,s_j \in S$. Strings outside this set, i.e. non-synchronising strings, are of special interest, and will be defined to be the set D(F):

$$D = D(F) = \{1 \in A^*: \mu(s_i,1) \neq \mu(s_j,1), \text{ for some } s_i,s_j \in S\}$$

For example, for the model in Figure 3.1, D = {Λ,0}, and for the model in Figure 3.2, D = {Λ,0,00,000, ... 0^k ... }.

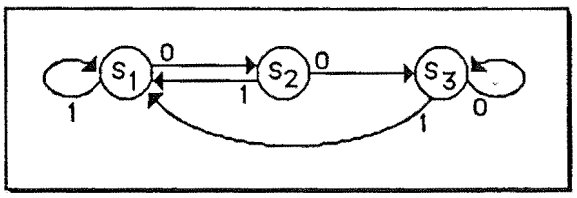


Figure 3.1: A Finite Context Automaton

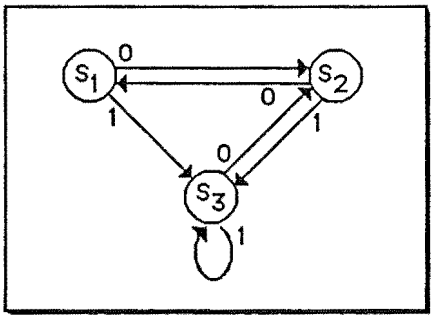


Figure 3.2: A Finite State Automaton

Define a *Finite Context Automaton* (FCA) to be an FSA, F, where D(F) is finite.

For an FSA, for any $m \notin D$, $\mu(s_i,m)$ is always the same, regardless of s_i, so it can be denoted as $\mu(m)$. For a particular state s, define its context to be the set of all strings which are in this manner guaranteed to result in transitions terminating at s, that is,

$$\text{context}(s) = \{m : m \notin D \text{ and } \mu(m) = s\}.$$

Define

$$c(s) = \text{context}(s) \cap (A.D - D),$$

where, informally, c(s) is a minimal set of suffixes which will result in transitions to state s from anywhere in the machine.

In Figure 3.1, $c(s_1) = \{1\}$, $c(s_2) = \{10\}$ and $c(s_3) = \{00\}$, and in Figure 3.2, $c(s_1) = \{100,10000,...1(00)^k...\}$, $c(s_2) = \{10,1000,...10(00)^k...\}$, $c(s_3) = \{1\}$.

From the above definitions, observe that for an FCA, where D is finite, that:

(a) If $m \in c(s)$ then $\mu(s,A^*m) = s$ i.e. $A^*m \in \text{context}(s)$

(b) c(s) is finite

(c) $\cup_i c(s_i) = A.D - D$

(d) $c(s_i) \cap c(s_j) = \varnothing$ for $s_i \neq s_j$

Observations (a) and (b) together imply that the current state is always determined by some finite size suffix of the input string, and that any information preceding this $c(s_i)$ will not be taken into account. In predictive models, each transition is assigned some probability, which is used to perform the prediction. Thus in an FCA, each character is predicted using some finite number of preceding symbols, and so implements a VOMM. An FCA used for prediction is a very pure form of VOMM because it does not use any escape characters to change contexts.

Observations (c) and (d) show that the context function partitions the set A^* into the context sets, and the finite set D.

Having defined FCAs, we now show that DMC generates only VOMMs by showing that every FSA generated is an FCA.

Chapter 4

Dynamic Markov Compression and Finite Context Automata

4.1 Dynamic Markov Compression

Cormack and Horspool describe the Dynamic Markov Compression (DMC) scheme, an adaptive data compression scheme which uses a Finite State Automaton (FSA) as the model [Cormack 87]. The scheme starts with a simple initial model, and adaptively adds states by "cloning" states under certain conditions. The result presented in this chapter is that, surprisingly, the DMC scheme does not use the full power of an FSA, but actually generates only FCAs, which have already been shown to be a form of variable-order Markov model.

In section 4.2, DMC (with its simplest initial model) is shown to generate only FCAs, by showing that the initial model is an FCA, and that cloning states in an FCA always produces another FCA. In section 4.3 the main theorem is extended to include all initial models proposed for DMC. This is done by viewing DMC models as having transitions on symbols rather than bits, where a symbol is a fixed size string of bits (typically 8-bit bytes).

Formal description of DMC

The DMC scheme uses an FSA, (S,B,μ,s), where $B = \{b_1,b_2, \cdots b_k\}$. For practical reasons, B is normally the binary alphabet, $\{0,1\}$. Each transition in the FSA has a probability assigned to it, based on how frequently it is used. The cloning function takes an FSA and a heavily used transition, $\mu(u,e)$, and creates a new state t', as shown in Figure 4.1 for $B = \{0,1\}$. The new state is created so that statistics for the heavily used transition

can be recorded separately, which should, in turn, lead to a better model of the input data and thus a more concise output representation.

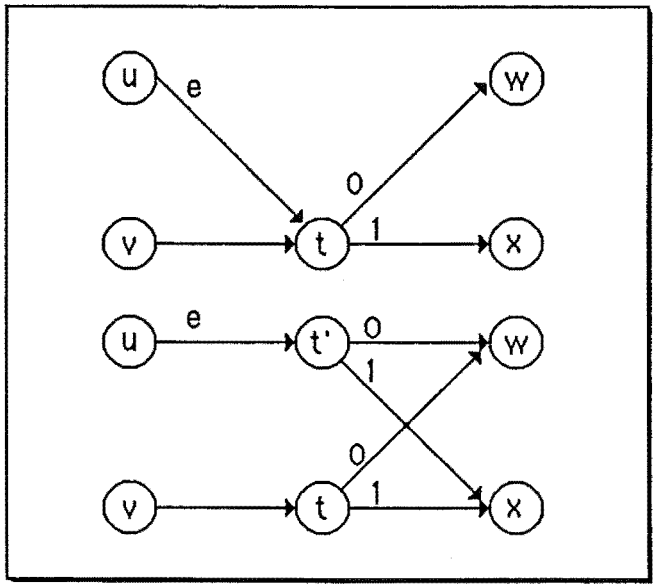


Figure 4.1: The DMC cloning function

Formally, the cloning function, δ, takes the FSA, (S,B,μ,s), a state/symbol pair (u,e), $u \in S, e \in B$, and generates a new FSA, $\delta((S,B,\mu,s),(u,e)) = (S',B,\mu',s)$, with

(1) $S' = S \cup \{t'\}$

(2) $\mu'(u,e) = t'$,

$\mu'(s,a) = \mu(s,a)$, $s \in S, a \in B, s \neq u$ or $a \neq e$,

$\mu'(t',a) = \mu'(t,a)$, $a \in B$.

In what follows, the concatenation of strings is extended to the concatenation of sets of strings. If $\{l_1,l_2,l_3 \ldots l_q\}$ is a set of q strings, and m is a string, then

$$\{l_1,l_2,l_3 \ldots l_q\}.m = \{l_1.m,l_2.m,l_3.m \ldots l_q.m\}.$$

If $\{m_1,m_2,m_3 \ldots m_r\}$ is a set of r strings, then

$$\{l_1,l_2,l_3 \ldots l_q\}.\{m_1,m_2,m_3 \ldots m_r\} = \{l_i.m_j : \forall\, i=1..q,j=1..r\}$$

Concatenation with the empty set yields the empty set.

4.2 Main Theorem

Before proving the main theorem, the following observations are made about the cloning function:

OBSERVATION 1

Transitions on strings in the cloned model which do not end at the state t', end up at the same state as they did before cloning, that is,

$$\text{for } s \in S,\ m \in B^*,\ \mu'(s,m) \neq t' \Rightarrow \mu'(s,m) = \mu(s,m).$$

This happens because on each transition, the μ' function is only different to the μ function when the transition is to t'. Because the new transitions out of t and t' are the same as the old transitions out of t, strings which pass through those states in the cloned model leave from them to the same state as they did in the original model.

OBSERVATION 2

For $s \in S$, $m \in B^*$, if $\mu(s,m) \neq \mu'(s,m)$ then $\mu(s,m) = t$ and $\mu'(s,m) = t'$. This follows from the converse of observation 1, given that $t' \notin S$.

LEMMA 1

Let l be a string of k symbols, and p be the first k-1 of these, i.e. $l = p.a$, $p \in B^*$, $a \in B$. If $\mu'(s_i,l) = t$, and $\mu'(s_j,l) = t'$, then $\mu(s_i,p) \neq \mu(s_j,p)$.

PROOF

Suppose $\mu(s_i,p) = \mu(s_j,p)$. Since the machine is deterministic and $\mu'(s_i,l) \neq \mu'(s_j,l)$, it must be that $\mu'(s_i,p) \neq \mu'(s_j,p)$. By observation 2, the only target which is different after cloning is t, which afterwards is t or t'. This implies that

$$\mu(s_i,p) = \mu(s_j,p) = t,$$

$$\text{and } \mu'(s_i,p) = t,\ \mu'(s_j,p) = t' \quad (\text{or } \mu'(s_i,p) = t',\ \mu'(s_j,p) = t),$$

which implies:

$t' = \mu'(s_j,p.a) = \mu'(\mu'(s_j,p),a) = \mu'(t',a) = \mu'(t,a) = \mu'(\mu'(s_i,p),a) = \mu'(s_i,p.a) = t$

which is a contradiction. A similar contradiction occurs if $\mu'(s_i,p) = t'$ and $\mu'(s_j,p) = t$, so Lemma 1 follows.

MAIN THEOREM

Every model generated by DMC when started on the initial "one-state Markov model" (Figure 4.2a), $F = (S,B,\mu,s_1)$, is an FCA, where $S = \{s_1\}$, $B = \{b_1,b_2...b_k\}$, $\mu(s_1,b_i) = s_1$.

PROOF

The proof is by induction. The initial model, F, has one state only, s_1, and every string leads to that state. This means that $D = \varnothing$, which is finite, so F is an FCA. The rest of the proof shows that cloning an FCA produces an FCA, by showing that if D is finite then D' is finite.

$D' = D((S',B,\mu',s_1))$

$= \{1 : \mu'(s_i,1) \neq \mu'(s_j,1),\ s_i,s_j \in S'\}$

By observation 2, the only strings with new targets are those with t or t' as their targets, and the empty string. Hence,

$D' = \{1 : \mu(s_i,1) \neq \mu(s_j,1),\ s_i,s_j \in S\} \cup \{1 : \mu'(s_i,1) = t,\ \mu'(s_j,1) = t',\ s_i,s_j \in S'\} \cup \{\Lambda\}$

The first term is the set D, and the middle term can be altered using Lemma 1, giving

$D' \subseteq D \cup \{1 : 1 = p.a,\ \mu(s_i,p) \neq \mu(s_j,p),\ s_i,s_j \in S, a \in B\} \cup \{\Lambda\}$

$\subseteq D \cup \cup_{a \in B}\{p : \mu(s_i,p) \neq \mu(s_j,p),\ s_i,s_j \in S\}.a \cup \{\Lambda\}$

$$= D \cup D.B \cup \{\Lambda\}$$

which is finite.

This then completes the proof.

4.3 Other initial models

Although the "one-state Markov model" (Figure 4.2a) is a suitable initial model, slightly better compression can be achieved by using more sophisticated initial models which correspond to bytes or words, rather than bits. In order to do this, and still retain the simple alphabet of B = {0,1}, these initial models contain cycles of h bits, typically with h=8. One of the structures suggested is a "tree", which is shown in Figure 4.2b, for h = 3. Also suggested is a "braid" structure, which is a generalisation of a tree, with h levels (typically 8), and 2^h nodes at each level (256). A given h-bit sequence follows transitions from any top level node down the braid, and back to a *unique* top level node.

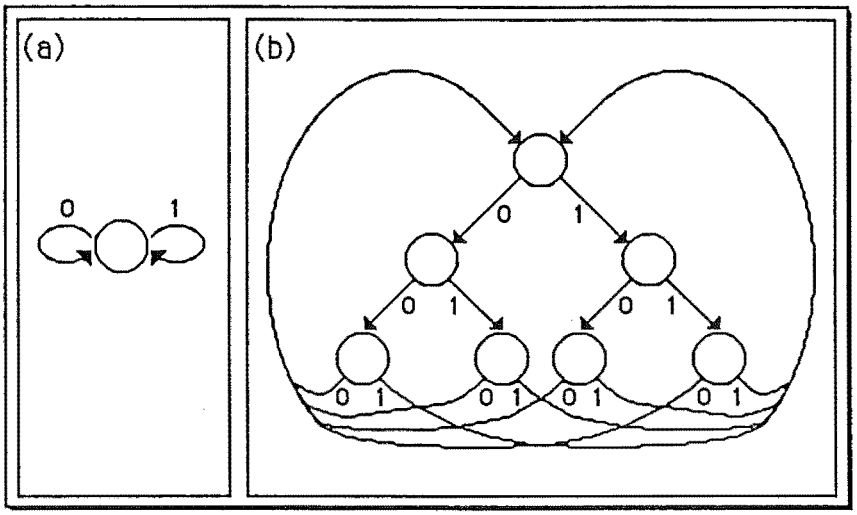


Figure 4.2: Initial models proposed for DMC:
(a) one-state Markov model (b) binary tree

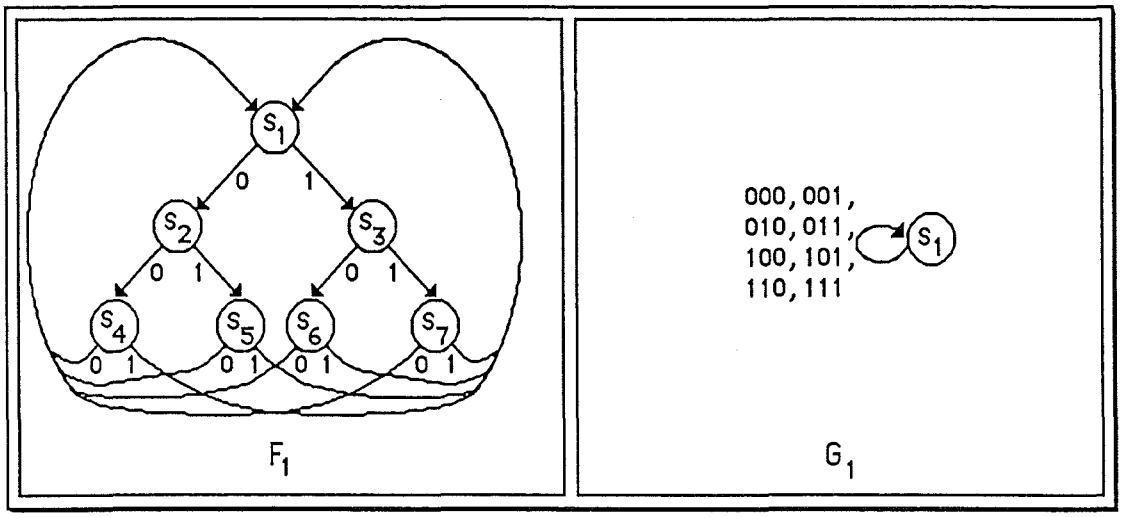


Figure 4.3: An initial binary tree of depth 3 (F_1), and its symbol level equivalent (G_1).

We are most interested in behaviour at the h-bit symbol level, where h is the length of cycles in the initial model. With this object, it is convenient to redefine the transition and cloning functions to allow for transitions drawn from the alphabet $A = B^h$, that is, bit strings of length h. Figure 4.3 shows an initial tree model, F_1, and its corresponding 3-bit symbol level equivalent. Viewing DMC models at the symbol level is possible because cloning only creates cycles with lengths which are multiples of h, so only nodes cloned from the starting state, s_1, are visited after each h bits of input. This can be verified by considering the effect of cloning a state which is part of a cycle (see Figure 4.1). Figure 4.4 shows how cloning s_1 is reflected in the symbol level equivalent, G_2, and how cloning other states has no effect on the equivalent (G_3). This new view ignores dependencies between bits, but retains the model structure as far as h-bit symbols are concerned.

The set of nodes cloned from s_1 is the set of all nodes visited from s_1 after a multiple of h bits:

$$S_h = \{s: \exists m \in A^*, \mu(s_1,m) = s\}$$

The symbol level transition function is

$$\mu_h : S_h \times A^* \rightarrow S_h,$$

$$\mu_h(s,1) = \mu(s,1), \quad s \in S_h, \; 1 \in A^*.$$

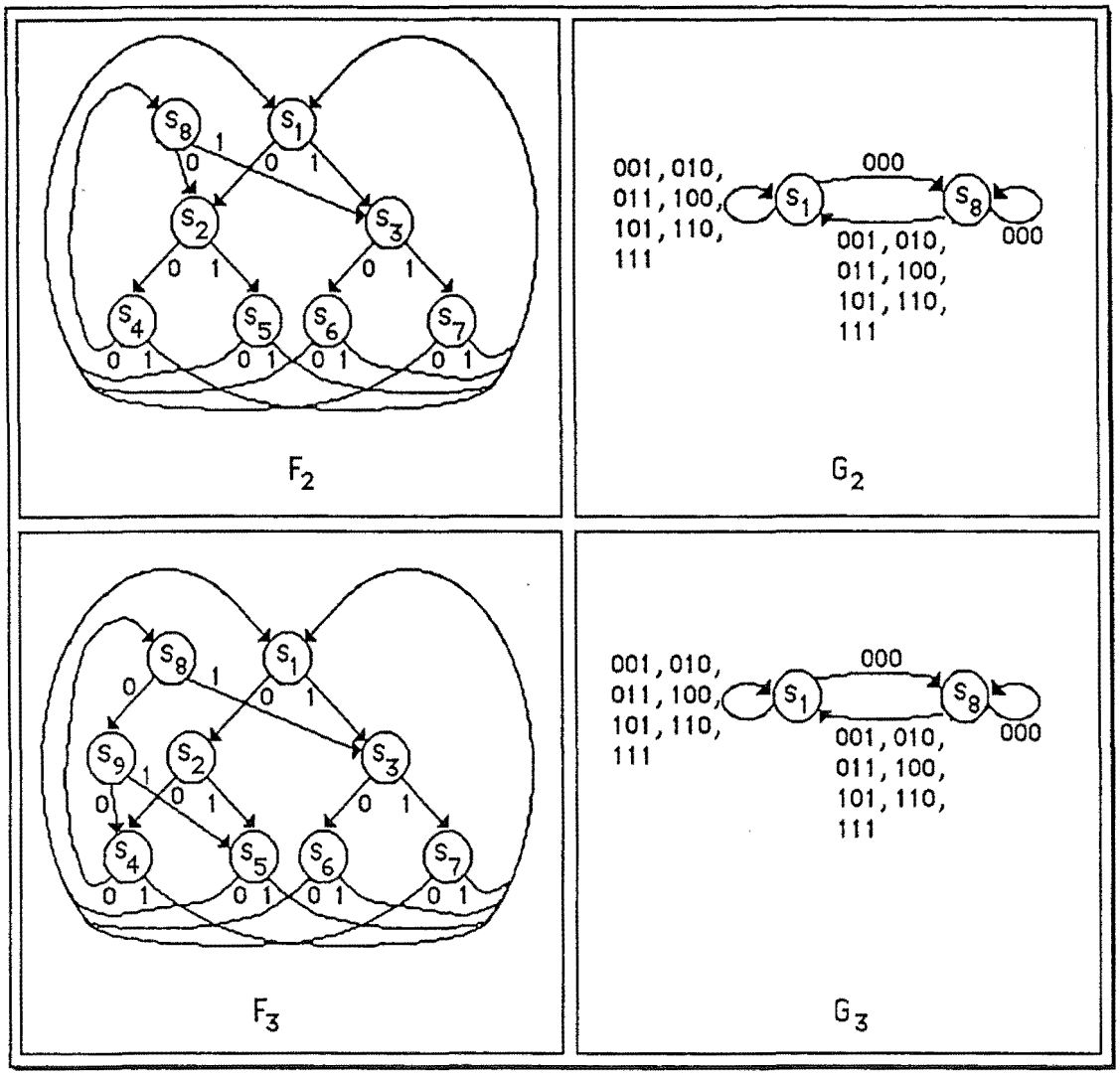


Figure 4.4: The effect of cloning on a symbol level equivalent. G_2 and G_3 are the equivalents of F_2 and F_3 respectively.

$F_2 = \delta(F_1,(s_4,0))$, $F_3 = \delta(F_2,(s_8,0))$, $G_2 = \delta_h(G_1,(s_1,000))$, $G_3 = G_2$.

A bit level cloning only results in a change at the symbol level when the cloning is on a transition which is the last bit of a symbol. Thus the bit-level cloning $\delta(u,e)$ transforms the symbol level model, F, to F', where

$F' = \delta_h(F,U)$ if $\mu(u,e) \in S_h$,

with $U = \{(s,a) : s \in S_h, a=a_1a_2...a_h, \mu(s,a_1a_2...a_{h-1})=u, a_h = e\}$

$F' = F$ otherwise

The new cloning function, $\delta_h(F,U)$, is very similar to δ:

$\delta_h((S_h,A,\mu_h,s_1),U) = (S_h',A,\mu_h',s_1)$, where

(1) $S_h' = S_h \cup \{t'\}$

(2) $\mu_h'(u,e) = t', \forall\ (u,e) \in U,$

$\mu_h'(s,a) = \mu_h(s,a),\ s \in S_h,\ (s,a) \notin U,$

$\mu_h'(t',a) = \mu_h'(t,a),\ a \in A.$

The two initial models suggested for DMC translate into two types of model at the symbol level, with the new alphabet $A = B^h$. The "tree" model corresponds to the zero-order Markov model:

$$F_0 = (S=\{s_1\},A,\mu_h,s_1),$$

$$\text{with}\quad \mu_h(s_1,a) = s_1, \forall\ a \in A.$$

The "braid" model corresponds to a first-order Markov model, with 2^h nodes, each node being labelled with one of the 2^h symbols in the alphabet:

$$F_1 = (S=\{s_a : a \in A\},A,\mu_h,s_1),$$

$$\text{with}\ \mu_h(s,a) = s_a, \forall\ a \in A, s \in S,$$

$$\text{and}\ \ s_1 \text{ is any } s_a \in S.$$

From this, $D(F_0) = \varnothing$, and $D(F_1) = \{\Lambda\}$, both of which are finite, so all the initial models are FCAs at the symbol level.

Because the symbol level view of the model has a cloning function isomorphic to those used in the proofs in section 4.2, the results of that section hold for the symbol level model. In particular, all models generated by DMC are FCAs at the symbol level, when started with any of the suggested initial models. For example, with an 8-bit initial model, every byte (character) is predicted using some finite number of preceding bytes. The probabilities used for prediction will be affected by interactions within the bit patterns of the symbols, but the overall probability used to encode any symbol is selected entirely by a Markov context.

The results in table 2.2 showed that DMC consistently gave slightly worse compression than the PPM scheme. It is possible that both the DMC and PPM schemes are approaching the optimal compression that can be achieved by a character-based variable-order Markov model. DMC has the advantage of being faster because each coding step requires a few array accesses and calculations, compared with a tree search used by PPM.

Chapter 5
Greedy Macro Encoding

5.1 Greedy Macro schemes

Storer and Szymanski [Storer 82] define a class of compression schemes called Macro encoding schemes, which "factor out duplicate occurrences of data, replacing the repeated elements with some sort of special marker identifying the data to be replaced at that point". Here, an important subset of Macro encoding schemes, called Greedy Macro (GM) encoding schemes, is defined.

5.1.1 Definition

A GM scheme is a compression scheme where, at each encoding step, a finite set of strings M is available, with each $m \in M$ allocated some code C(m). The encoder searches for the longest string, $s \in M$, which matches the next characters in the text to be encoded, and uses C(s) to encode them.

For example, if M = {a,b,ba,bb,abb}, and
C(a) = 00, C(b) = 010, C(ba) = 0110, C(bb) = 0111 and C(abb) = 1,
then "babb" is coded in 8 bits as:
C(ba).C(bb) = 0110.0111

5.1.2 Comments

Notice that greedy parsing is not necessarily optimal. For example, the string of the previous example could have been coded in 4 bits as
C(b).C(abb) = 010.1

However, determining an optimal parsing is difficult in practice, because there is no limit to how far ahead the encoder may have to look. This is illustrated by coding the string with prefix $(ba)^i b$ (i.e. bababa ··· bab) using M={a,b,ab,ba,bb}, and

C(a) = 000, C(b) = 001, C(ab) = 10, C(ba) = 11, C(bb) = 0100000.

If the string being encoded is followed by the character 'a' then the optimal parsing is

ba,ba,ba, ··· ,ba, ba,

but if the next character is 'b', the optimal parsing would be

b, ab, ab, ab, ··· ,ab, b.

To ensure optimal encoding in this situation, the encoder must look ahead $2i + 1$ characters, where i can be arbitrarily large. The greedy approach is not optimal, but it is used widely in practical schemes because it allows single pass encoding with a bounded delay. The furthest a GM scheme needs to look ahead is the size of the longest string in M.

5.2 Codes, lengths, and code space

GM schemes map a set of strings, M, onto a set of codes, using the function C(m), $m \in M$. Define the function L(m) to be the length of C(m) in bits :

$$L(m) = |C(m)|$$

The amount of code space allocated to m is then

$$R(m) = 2^{-L(m)}$$

These three functions are extended recursively to the set of strings in M*. If the greedy algorithm parses the prefix m from the string m.q, $m \in M$, $q \in M^*$, define

$C(\Lambda) = \Lambda$, $C(m.\Lambda) = C(m)$, $C(m.q) = C(m).C(q)$ (concatenation)

$L(\Lambda) = 0$, $L(m.\Lambda) = L(m)$, $L(m.q) = L(m) + L(q)$ (addition)

$R(\Lambda) = 1$, $R(m.\Lambda) = R(m)$, $R(m.q) = R(m).R(q)$ (multiplication)

In what follows it will be assumed that the GM scheme fully utilises the code space i.e. $\Sigma_{m \in M} R(m) = 1$. For a scheme where $\Sigma_{m \in M} R(m) < 1$, an extra artificial phrase could be added to M, to bring the code space used to unity. This will not alter the code space used

by the scheme because the artificial phrase will never be used during coding, but it simplifies the following descriptions.

5.3 Identification of GM schemes in the literature

In this section, the set M, and the functions L(m) and/or R(m) will be identified for each GM scheme described in chapter 2. The allocation of C(m) is arbitrary, provided that |C(m)| = L(m). The fundamental difference between VOMM schemes and GM schemes is that VOMM encoding is based on the characters to the *left* of the encoding position, while GM encoding works with the characters to the *right* of the encoding position. Zero-order Markov modelling lies at the intersection of the two approaches, and so three zero-order schemes (ASCII, Huffman, and Macwrite coding) appear here with the GM schemes.

In the following, A is the set of 128 ASCII characters.

ASCII: M = A

L(m) = 8

Huffman: M = A

L(m) is assigned by Huffman's algorithm

Macwrite: M = A

$$L(m) = \begin{cases} 4 & \text{if } m \in \{\text{<space>},e,t,n,r,o,a,i,s,d,l,h,c,f,p\} \\ 12 & \text{otherwise} \end{cases}$$

Digram: $M = A \cup$

$\{\text{<space>},a,e,i,o,n,t,u\} \times \{\text{<space>},e,t,a,o,n,r,i,s,h,d,l,f,c,m,u\}$,

where $S \times T$ is the set of all digrams formed by taking the first character from S and the second from T.

L(m) = 8

Pike: M = {<space>,<eoln>,e,a,i ...,c} ∪ {u,p, ...,',',} ∪
{"the","of", ...,"on"} ∪ {z,A,B ... ,"pay"}

$$L(m) = \begin{cases} 4 & \text{if } m \in \{\text{<space>,<eoln>,e,a,i ..c}\} \\ 8 & \text{if } m \in \{\text{u,p, ...,',',}\} \cup \{\text{"the","of", ...,"on"}\} \\ 12 & \text{otherwise} \end{cases}$$

LZ77: LZ77 alternates between two GM schemes :

(1) Pointers:

M = {every string in the previous N-F characters of length 0 to F-1}

$L(m) = \lceil \log_2 N \rceil + \lceil \log_2 F \rceil$

$R(m) \approx 1/NF$

(2) Characters

$M = A$

$L(m) = \log_2 |A|$

LZSS: $M = A \cup$ {every string in the previous N-F characters of length p to p+F-1}

$$L(m) = \begin{cases} \log_2 |A| + 1 & \text{if } m \in A \\ 1 + \lceil \log_2 N \rceil + \lceil \log_2 F \rceil & \text{otherwise} \end{cases}$$

$$R(m) = \begin{cases} 1/2 * 1/|A| & \text{if } m \in A \\ \approx 1/2 * 1/NF & \text{otherwise} \end{cases}$$

LZ78: LZ78 alternates between two GM schemes :

(1) Pointers:

M = {all phrases parsed so far}, $|M| = \rho$,

$L(m) = \lceil \log_2 \rho \rceil$

$R(m) \approx 1/\rho$

(2) Characters

$M = A$

$L(m) = \log_2 |A|$

LZW: $M = A \cup \{\text{the first } 4096 - |A| \text{ phrases parsed}\}$, $|M| \leq 4096$,

$L(m) = \log_2 4096 = 12$ bits

LZC: $M = A \cup \{\text{the current list of parsed phrases}\}$, $|M| = \rho$,

$L(m) = \lceil \log_2 \rho \rceil$

$R(m) \approx 1/\rho$

LZJ: $M = A \cup \{$every *unique* string of length 1 to h in the previous text$\}$,

$|M| \leq H$.

The pruning operation removes strings which have occurred only once.

$R(m) = 1/H$

$L(m) = \log_2 H$

5.4 Decomposition

Decomposition is a process where the code space assigned to a phrase is divided into the code space used by each character of the phrase. It is sometimes called the *symbolwise equivalent*. A decomposed model can be used with arithmetic coding to code text in exactly the same number of bits as the original model. For example, if the phrase "ab" is allocated 1/8 of the code space (i.e. coded in 3 bits), one possible decomposition is to allocate 1/4 of the code space (2 bits) to a character 'a' in the empty context, and 1/2 (1 bit) to a 'b' following an 'a'. The total code space allocated to "ab" is obtained by multiplying the individual code spaces (adding the individual lengths). Although a decomposition need not be unique, it may be constrained by the codes allocated to other phrases. In the next example, all phrases of a GM scheme are decomposed. The set M has been chosen so that only one decomposition is possible.

Example 1

The decomposition of the GM scheme with A = {a,b}, M= {a,ba,bb}, and

L(a) = 2, L(ba) = 2, L(bb) = 1,

R(a) = 1/4, R(ba) = 1/4, R(bb) = 1/2,

is show in Figure 5.1, giving the following zero/first-order VOMM, where P(x|s) is the code space allocated to the character x when preceded by the string s:

P(a|Λ) = 1/4, P(b|Λ) = 3/4, P(a|b) = 1/3, P(b|b) = 2/3

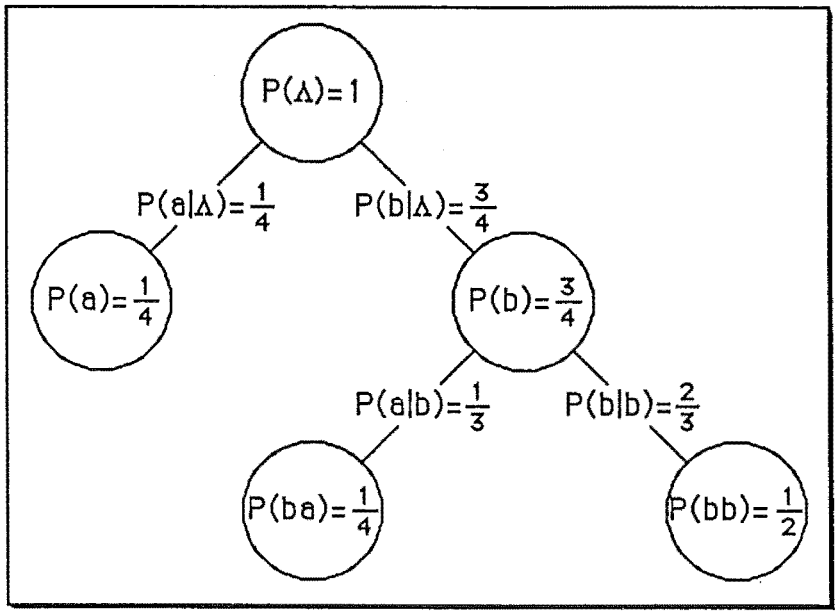


Figure 5.1 : A simple decomposition

In the decomposition, an 'a' is coded in -log 1/4 = 2 bits, as before, but a 'b' is coded in -log 3/4 = 0.415 bits. However, after a 'b', 'a' is coded in -log 1/3 = 1.585 bits, and a 'b' is coded in -log 2/3 = 0.585 bits. Thus the pair 'ba' still uses a total of 2 bits, and 'bb' uses 1 bit, but the encoder *did not have to look ahead* to determine which encoding to use.

A decomposed model will not necessarily be of practical use, but it can be used to gain insight into how a scheme achieves compression.

5.5 Decomposition techniques in the literature

The principle of decomposition was first introduced by Shannon [Shannon 48]. Rissanen and Langdon [Rissanen 81] give an algorithm for decomposing GM schemes, with restrictions on the set M. One important restriction is that *no* string in M is allowed to be the prefix of any other string in M.

The decomposition was performed by inserting every string in M into a trie (as in Figure 5.1), and assigning a code space P(s) to each node of the trie, s, where P(s) is the sum of the code space assigned to sons of s. If s is a leaf node, it is assigned code space R(s). Each transition is given code space P(x|s) = P(s.x)/P(s). The trie is made into an FSA by changing transitions to leaves of the tree so that they go to the root instead (Figure 5.2).

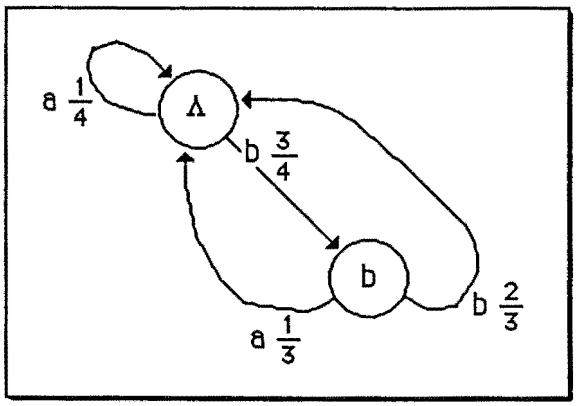


Figure 5.2 : Langdon and Rissanen's decomposition

Langdon [Langdon 83a] gives a decomposition for LZ78, which can easily be extended to GM schemes with the restriction that the prefix of *every* string in M is also in M. Langdon's construction also uses a trie, but to construct an FSA, missing transitions (not necessarily from leaves), were taken to the root using a special *escape* character (called a *comma* by Langdon). This escape character marks the unexpected end of a parsing; in the FSA it corresponds to resetting the Markov context to the empty string.

Some of the GM schemes do not conform to the above restrictions. For example, the set M for Pike's scheme includes "the" and "there", but not "ther". In the following section, a construction is given for an FSA which represents the decomposition of any GM scheme, with no restrictions on the set M. The construction is similar to the previous two approaches, but some escape transitions will be linked within the trie, rather than to the root. The behaviour of the FSA will be shown to be equivalent to a VOMM.

Note that the construction given here does not allow for the set M to change during encoding, except under special circumstances. This means that for the LZ schemes, where the set M is not static, the existence of a VOMM equivalent can only be shown for each

coding step, and not for the coding of a whole text. Whether the general FSA can be modified to accommodate changes "on the fly" is still unknown, although it can been done for the specific cases of LZ78 [Langdon 83a] and LZ77 (chapter 6). The construction given here also leaves open the problem of switching between different models, but the only scheme which requires this is LZ77, for which a specific solution is given in chapter 6.

5.6 Decomposing a GM scheme

The following GM scheme will be used to illustrate the construction of a VOMM equivalent to any GM scheme. It has been chosen because it cannot be decomposed using Rissanen and Langdon's techniques.

Example 2

M= {a,b,aab},

C(a) = 01, C(b) = 00, C(aab) = 1,

L(a) = 2, L(b) = 2, L(aab) = 1,

R(a) = 1/4, R(b) = 1/4, R(aab) = 1/2.

The basis of the FSA, F = (S,A,μ,Λ), is a directed trie which is constructed by adding transitions from the root for each s$\in$M (Figure 5.3). Each state is labelled by its path from the root, the root itself being labelled "Λ". At this stage S contains every prefix of strings in M, including the strings themselves. The label of each state will be the context of that state when the FSA has been constructed.

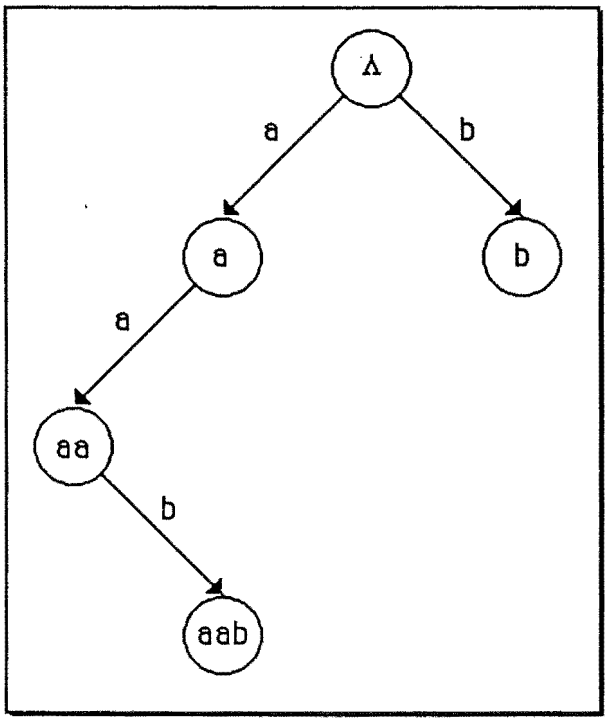


Figure 5.3: Directed trie for example 2

For each non-leaf node s∈ S, with a transition on x∈ A not yet defined, add a new leaf node, s.x, to S, and the transition μ(s,x) = s.x (Figure 5.4).

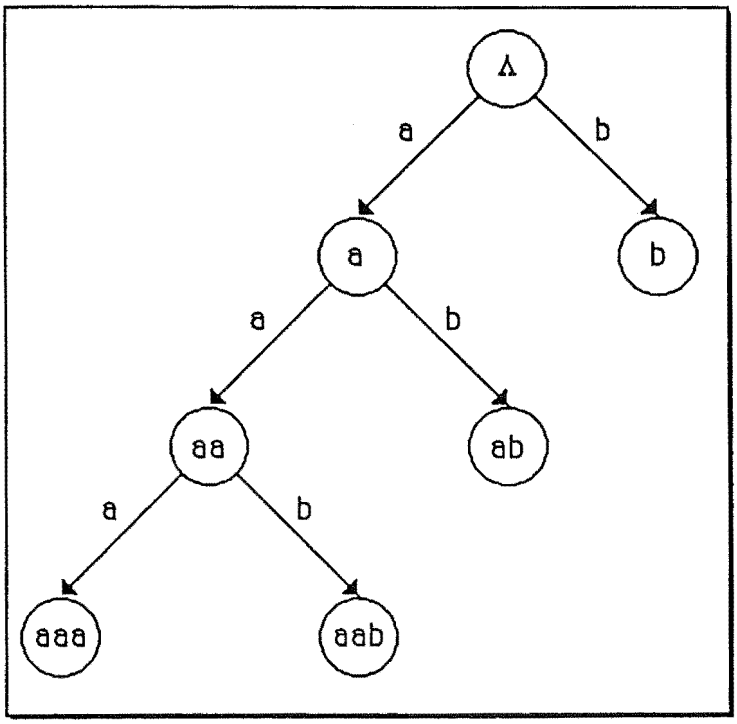


Figure 5.4: Addition of transitions from non-leaf nodes

The next step is to add the escape transitions. For each leaf node, $s=s_1s_2\cdots s_n$, define $u=s_1s_2\cdots s_i$ to be the longest prefix of s uniquely parsed by the greedy algorithm. The choice of leaf nodes ensures that $|u|>0$, because a leaf node is never a *proper prefix* of a string in M. Let t be the remaining part of s i.e. $t = s_{i+1}s_{i+2}\cdots s_n$, so s = u.t. Add the escape transition $\mu(s,')=t$ to each non-leaf node (Figure 5.5).

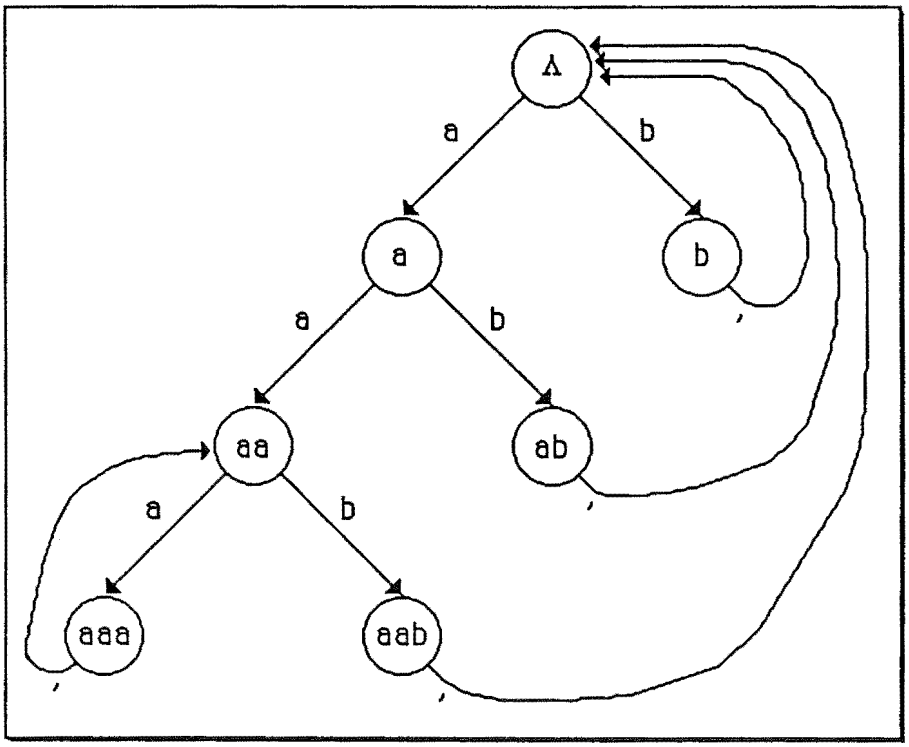


Figure 5.5: Addition of escape (comma) transitions

Each node, s, is assigned code space P(s), to allow for each transition out of s (Figure 5.6). P(s) is defined to be the code space assigned by the function R to all infinitely long strings from M* with s as a prefix. Although the sum is infinite, it can be evaluated in a finite amount of time, as shown by the following example.

Consider the evaluation of P(aa) for the GM scheme of example 2. The only infinite strings in M* with "aa" as a prefix are those of the form "aab,v_k", "a,aab,v_k", or "a,a,v_k", where v_k is any string in M*. The total code space allocated to these three forms is obtained by multiplying the code space of the phrases in a string, and summing over each possible string, giving

$P(aa) = R(aab)\Sigma_k R(v_k) + R(a)R(aab)\Sigma_k R(v_k) + R(a)R(a)\Sigma_k R(v_k)$.

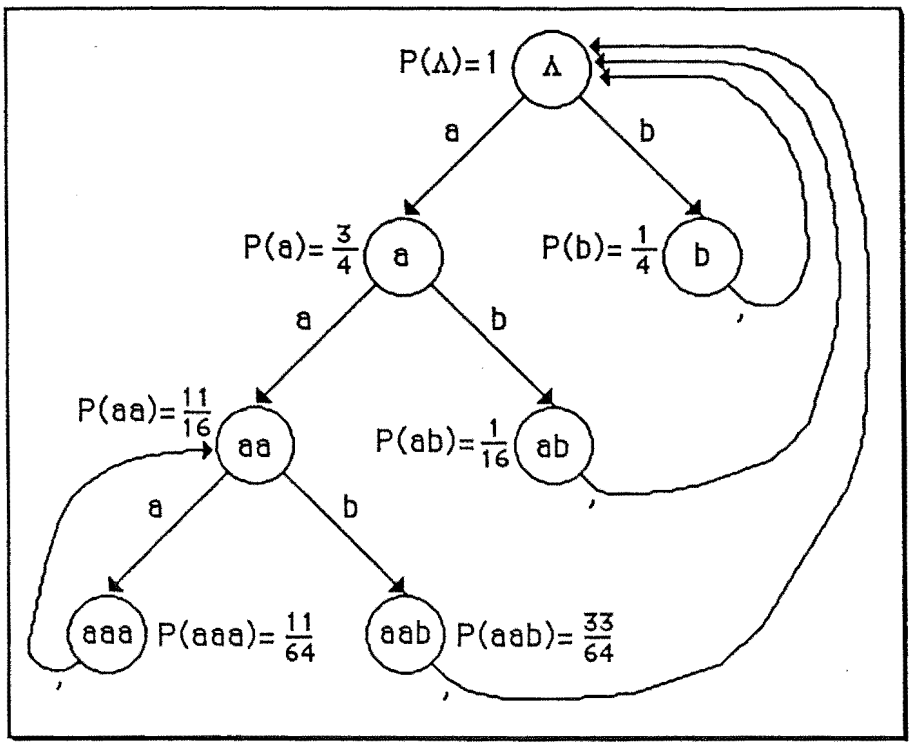


Figure 5.6: Allocation of code space to nodes

Because $\Sigma_k R(v_k)$ is the code space allocated to all strings, it is unity, and so

$P(aa) = R(aab) + R(a)R(aab) + R(a)R(a) = 1/2 + 1/4*1/2 + 1/4*1/4 = 11/16.$

For the general case, the following recursive function calculates P(s) in finite time:

```
function P(s):
  P := 0
  for each m∈M do
    if s is a prefix of m (including s=m), m=s.q,
      P := P + R(m)
    else if m is a prefix of s, s=m.q,
      P := P + R(m).P(q)
```

P(s) allows space for every string beginning with s which might follow. By definition, $P(\Lambda) = \Sigma_{m \in M} R(m) = 1$.

Every non-leaf node $s \in S$ has a transition for every $x \in A$, $\mu(s,x) = s.x$. Each of these transitions is assigned a code space using

$$P(x|s) = P(s.x)/P(s).$$

Every leaf node s∈S has only one outgoing transition, μ(s,',') = t, and this transition is assigned the codespace

$$P(',|s) = R(u)P(t)/P(s),$$

where s = u.t.

The FSA is now complete (Figure 5.7).

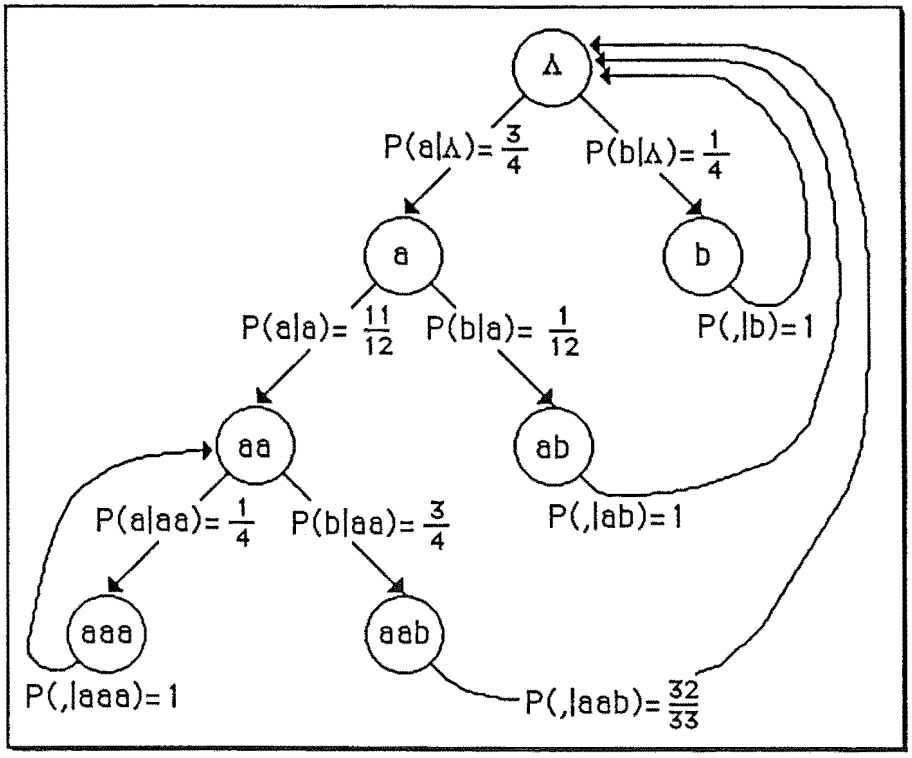


Figure 5.7: The final FSA with transition probabilities

Observations

The machine is equivalent to a VOMM because for each input character it moves down the tree, determining a longer context for encoding the next character. The context must be finite because it cannot be longer than the longest string in M. The escape character resets the machine to a shorter context when a leaf is reached i.e. when no longer context can be found in the tree.

Figure 5.7 illustrates how the task of looking ahead to determine the encoding is implemented in a symbol-wise approach. By following the action of the FSA with the

string "aaaaaaab", we see that after the first two 'a's are encountered, the FSA cycles on subsequent 'a's, giving each 'a' a code space of 1/4 (2 bits) until the 'b' is encountered. Thus the FSA encoder associates the first two 'a's with the final 'b' to encode the phrase "aab".

Notice that the transitions out of each node do not necessarily use all of the code space i.e. do not add up to 1. This is because the original scheme may contain redundant codes. For example, the scheme of example 2 will never code the sequence "aab" as C(a).C(a).C(b), which accounts for the unused code space out of node "aab" in the FCA. Also, the decomposition is not unique. The FCA in Figure 5.7 would still be equivalent if P(a|a) = 8/9, and P(',|aab) = 1.

To show that the code space used by the FSA is the same as that used by the original GM scheme, consider the coding of the string $s=s_1s_2\cdots s_n$. The machine follows transitions down the tree from the root until a leaf node is reached, say at character s_j. The transitions followed are:

$$\mu(\Lambda,s_1)=s_1,\ \mu(s_1,s_2)=s_1s_2,\ \mu(s_1s_2,s_3)=s_1s_2s_3,\ \cdots\ \mu(s_1\cdots s_{j-1},s_j) = s_1\cdots s_j.$$

At this point the transition $\mu(s_1\cdots s_j,',')=t$ is taken, where s=u.t, and u is the longest prefix of $s_1\cdots s_j$ parsed by the greedy algorithm. Let $u = s_1\cdots s_i$ and $t=s_{i+1}\cdots s_j$. Recall that the choice of leaf nodes ensures that |u|>0. The code space used by the transitions up to now is

$$P(s_1|\Lambda)P(s_2|s_1)P(s_3|s_1s_2)\cdots P(s_j|s_1\cdots s_{j-1})P(',|s_1\cdots s_j)$$

$$= \frac{P(s_1)}{P(\Lambda)}\cdot\frac{P(s_1s_2)}{P(s_1)}\cdot\frac{P(s_1s_2s_3)}{P(s_1s_2)}\cdot\ \cdots\ \frac{P(s_1\cdots s_j)}{P(s_1\cdots s_{j-1})}\cdot\frac{R(u)P(t)}{P(s_1\cdots s_j)}$$

$$= R(u)P(t)$$

To verify that this is the situation desired, consider the action of the FSA encoding the string s, but with the prefix u removed i.e. the string $s_{i+1}\cdots s_j\cdots s_n$. Immediately after the

character s_j is encountered, the machine would be in state t ($=s_{i+1}\cdots s_j$), and the code space used would be:

$$\frac{P(s_{i+1})}{P(\Lambda)} \cdot \frac{P(s_{i+1}s_{i+2})}{P(s_{i+1})} \cdot \cdots \frac{P(s_{i+1}\cdots s_j)}{P(s_{i+1}\cdots s_{j-1})} = P(t).$$

So the machine will be in the same state whether it had coded $s=s_1\cdots s_n$ or $s_{j+1}\cdots s_n$, but in the former case has used the extra code space R(u), which is the same as the code space allocated by the GM scheme to the prefix u. Encoding proceeds for the remainder of s in a similar manner, parsing off prefixes and moving to an appropriate state to allow for the delay in recognising parsed phrases.

Chapter 6
Ziv-Lempel coding

Although a general construction was given in chapter 5 for a symbol-wise equivalent for the class of Greedy Macro encoders, of which LZ77 is a member, it is helpful to look at a specific symbol-wise equivalent for the LZ77 scheme. The insight gained from this symbol-wise equivalent has revealed inefficiencies in the coding, leading to improvements in the LZSS scheme which achieve better compression, and make the choice of the size of the window and lookahead buffer less critical.

6.1 The symbol-wise equivalent of LZ77

6.1.1 Outline

The following symbol-wise model generates probabilities for an arithmetic coder, and will be shown to be equivalent to an LZ77 encoder coding characters from an alphabet A of |A| characters. The reader should bear in mind that the scheme is designed to be give exactly the same amount of compression as LZ77, and to achieve this it will contain obvious inefficiencies. The LZ77 encoder has a window of N characters, and a maximum match length of F characters, as described in section 2.6.4.

At each coding step in the symbol-wise model, the N-F characters directly preceding the coding position are used as a sample to calculate the current relative frequency of a character, which is used as an estimate for its probability. The sample size will be denoted by M = N-F. The estimated probability of a character is the number of times it occurs in its context, divided by the number of times the context occurs. The initial context for coding is the empty string (zero-order Markov context). The idea is to code character by character, growing the context as each character is coded, until the context is unusable because of a

zero probability, or until the context reaches the maximum length of F characters. When this happens, the context is reset to the empty string, and the growing process is repeated.

For each input character there are at least two output messages. The first gives the character, and the second is a YES/NO message to indicate if the character is valid. An invalid character occurs when its probability is zero, or when the context has grown to F characters. This condition requires an extra message to identify the character. After this message, the encoder and decoder reset the context to the empty string. The details of the algorithm follow:

If the coding position is character a_i, then the sample for calculating probabilities is the string $a_{i-M} \cdots a_{i-1}$. If $j \leq 0$ then assume that a_j is a blank. Define c(s) to be the number of times the string s occurs in the sample. For example, $c(a_i a_{i+1})$ is the count of the strings of length two in the sample containing a_i followed by a_{i+1}. For the empty string, denoted by Λ, $c(\Lambda)=M$. These counts are used to estimate probabilities conditioned by Markov contexts. Let P(a|s) be the probability of the character a occurring in the context of string s . If s.a is the concatenation of s and a then this probability is estimated from the sample using P(a|s) = c(s.a)/c(s) i.e.

$$P(a_i) = \frac{c(a_i)}{c(\Lambda)}$$

$$P(a_{i+1}|a_i) = \frac{c(a_i a_{i+1})}{c(a_i)}$$

$$P(a_{i+2}|a_i a_{i+1}) = \frac{c(a_i a_{i+1} a_{i+2})}{c(a_i a_{i+1})}$$

$$P(a_{i+F-1}|a_i a_{i+1} \ldots a_{i+F-2}) = \frac{c(a_i a_{i+1} \ldots a_{i+F-1})}{c(a_i a_{i+1} \ldots a_{i+F-2})}$$

6.1.2 Encoding algorithm

The following algorithm generates probabilities for an arithmetic coder, and is equivalent to LZ77.

```
integer i; {the coding position}
integer x; {the number of characters in the context}

i:=1;        {start encoding at the first character}
x:=0;        {context is the empty string}

while not eof do
   if c(a_i a_{i+1} ... a_{i+x}) ≠ 0  and  x ≠ F then
      begin
      encode a_{i+x} using P(a_{i+x} | a_i a_{i+1} ... a_{i+x-1});

                                                          F-x-1
      encode "valid character" using P("valid character") = ------;
                                                           F-x

      x := x + 1      {increase the context by 1 character}
      end
   else
      begin
                                                              1
      encode some invalid character, χ , using P(χ) = ----------------------;
                                                      c(a_i a_{i+1} ... a_{i+x-1})
                                                           1
      encode "invalid char" using P("invalid character") = ------;
                                                           F-x
      encode character a_{i+x} using P(a_{i+x}) = 1/|A|;
      i := i + x + 1;    {move the coding position}
      x := 0             {reset context to empty string}
      end;
```

In this algorithm, the probability of a character is calculated using the increasing context (if possible). The probability of the "valid character" and "invalid character" messages are calculated using the strategy:

> "The invalid character is equally likely to occur at each position between here and the last possible place for a valid character (a_{i+F-1})".

6.1.3 Analysis

The effect of finding contexts in this way is to break the text up into *chunks*, where a chunk is the longest phrase that has occurred in the previous M characters, plus one character. This is the same as one LZ77 pointer/character pair.

The number of bits used by a chunk which codes y+1 characters is the sum of three components: the character codes, the "valid/invalid character" messages, and the explicit characters. These are analysed separately. It is assumed that an event, e, with probability P(e) will be coded in -log P(e) bits, where logarithms are in base 2.

(1) *Bits used for the character codes*

$$= -\log P(a_i) - \log P(a_{i+1}|a_i) \cdots - \log P(a_{i+y-1}|a_i a_{i+1} \ldots a_{i+y-2}) - \log \frac{1}{c(a_i \ldots a_{i+y-1})}$$

$$= -\log \frac{c(a_i)}{c(\Lambda)} - \log \frac{c(a_i a_{i+1})}{c(a_i)} - \log \frac{c(a_i a_{i+1} a_{i+2})}{c(a_i a_{i+1})} \cdots - \log \frac{c(a_i \ldots a_{i+y-1})}{c(a_i \ldots a_{i+y-2})} - \log \frac{1}{c(a_i \ldots a_{i+y-1})}$$

$$= -\log \frac{c(a_i)}{c(\Lambda)} \cdot \frac{c(a_i a_{i+1})}{c(a_i)} \cdot \frac{c(a_i a_{i+1} a_{i+2})}{c(a_i a_{i+1})} \cdots \cdot \frac{c(a_i \ldots a_{i+y-1})}{c(a_i \ldots a_{i+y-2})} \cdot \frac{1}{c(a_i \ldots a_{i+y-1})}$$

$$= -\log \frac{1}{c(\Lambda)}$$

$$= \log M$$

$$= \log (N-F)$$

(2) *Bits used for "valid/invalid character" messages* (characters a_i to a_{i+y-1} are valid, character a_{i+y} is invalid):

$$= -\log \frac{F-1}{F} - \log \frac{F-2}{F-1} - \log \frac{F-3}{F-2} \ldots - \log \frac{F-y}{F-y+1} - \log \frac{1}{F-y}$$

$$= -\log \frac{F-1}{F} \cdot \frac{F-2}{F-1} \cdot \frac{F-3}{F-2} \cdots \frac{F-y}{F-y+1} \cdot \frac{1}{F-y}$$

$$= \log F$$

(3) *Bits used for explicit character* = -log 1/|A| = log |A| (= 7 for ASCII codes).

The analysis shows that phrases and explicit characters in the symbol-wise equivalent correspond to the phrases and explicit characters in the LZ77 scheme.

6.2 Observations

The symbol-wise approach shows that hidden behind the LZ77 scheme is a variable-order Markov model, similar to that of Cleary and Witten's PPM scheme [Cleary 84b]. The Markov model used for predicting the next character starts with a zero-order context, and increases the context by one for each character encoded, until the context is unusable because it does not occur in the sample. At this point an explicit character is transmitted and the context is reset to zero-order. In contrast, the PPM scheme begins with a large context for each character, and reduces it until a usable size is reached. If no usable context occurs, an explicit character is used. The results in Table 2.2 show that PPM achieves significantly better compression than LZ77.

Table 6.1 shows further results from experiments with LZ77. They are the average number of characters represented by each pointer, and the average size of the Markov context used in the symbol-wise equivalent for five of the benchmark texts.

File	F yielding best CR	Average match length	Average context used	CR(%)
matthew	15	5.0	3.0	49.7
short	15	3.6	2.5	65.4
csh	15	5.3	3.3	47.5
lzss	63	9.6	13.4	30.7
session	63	10.4	16.2	28.4

Table 6.1: Measurements from LZ77 with N = 8192 and F chosen for the best CR.

Table 6.1 shows that a larger average context size always corresponded to a better CR. The principal advantage of LZ77 over symbol-wise schemes, such as PPM and the LZ77 symbol-wise equivalent, is that decoding is very fast, and requires very little memory [Bell 86]. This is achieved at the hidden cost of inefficiencies in the use of the available information (such as transmitting an invalid character for each phrase). Thus any improvement in compression achieved in the light of the following observations is only worthwhile if it does not add significantly to the task of decoding. The improvements to LZSS in the following section are evaluated using empirical results from the benchmark texts. Thus the scheme is "fine-tuned" on a small sample of text, and not necessarily useful for other texts, or other types of data. In section 6.4, the improvements are evaluated on a large number of texts to verify their usefulness.

6.3 Improvements to LZSS

The output of an LZSS file is made up of characters and pointers. Each character is made up of a flag bit and a 7-bit ASCII code. Each pointer is made up of a flag bit, a *reach* (locates the target of the pointer) and a *cover* (the number of characters represented by the pointer). Thus there are four different components: flags, explicit characters, reaches, and covers. In considering improvements to LZSS, methods for coding each component are now investigated. Figure 6.1 shows how much each component contributed to the compression ratio for each benchmark file, and hence the relative importance of reducing the space used by each component.

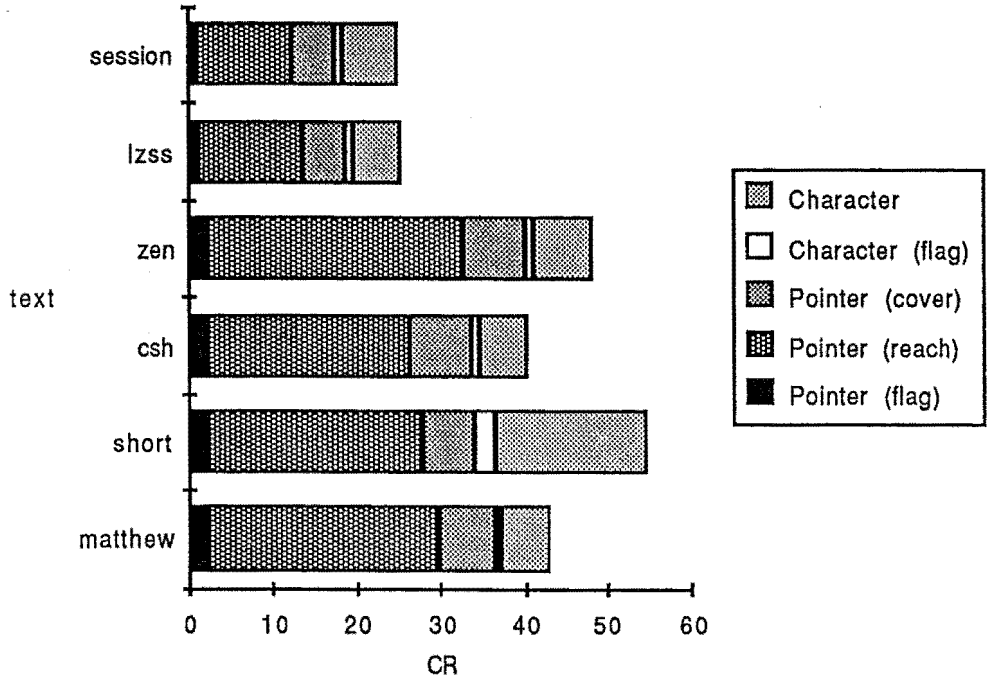


Figure 6.1: Components of the LZSS CR.

6.3.1 Pointer covers

In the symbol-wise model, the probability of the "valid/invalid character" messages implies that all match lengths are equiprobable, which is not the case in practice. This could be changed by introducing some variable length coding for match lengths. Although arithmetic coding and Huffman coding could generate codes which correctly match the probabilities of different match lengths, they would add considerably to the complexity of the decoder, and would ideally require two pass encoding. To maintain simplicity of decoding, several variable length codings of the integers were investigated and compared with arithmetic coding. The codings are detailed in appendix C. The code C_α (unary coding) corresponds to using p("valid character") = 1/2 in the symbol-wise model of LZ77. The normal (binary) coding of the match length used by LZ77 and LZSS is Elias' C_β.

Because arbitrarily large numbers can be represented by the codings evaluated, the maximum match length (F) can be set to a value much larger than any actual match length, without any loss of compression. Thus the parameter F is effectively eliminated. The encoder's task of finding a longest match when F is very large is handled well by the binary tree algorithm in chapter 7. The amount of memory required by this algorithm is not affected by F, and search time increases very slowly as F increases.

In order to evaluate the effectiveness of each coding method proposed for covers, the following experiment was performed. LZSS was implemented with a very large value for the parameter F, and for each cover of size i, its number of occurrences, N(i), was recorded. For each coding of the integers, C(i), the number of bits used to code the covers is calculated as

$$\Sigma\, N(i)\, |C(i)|$$

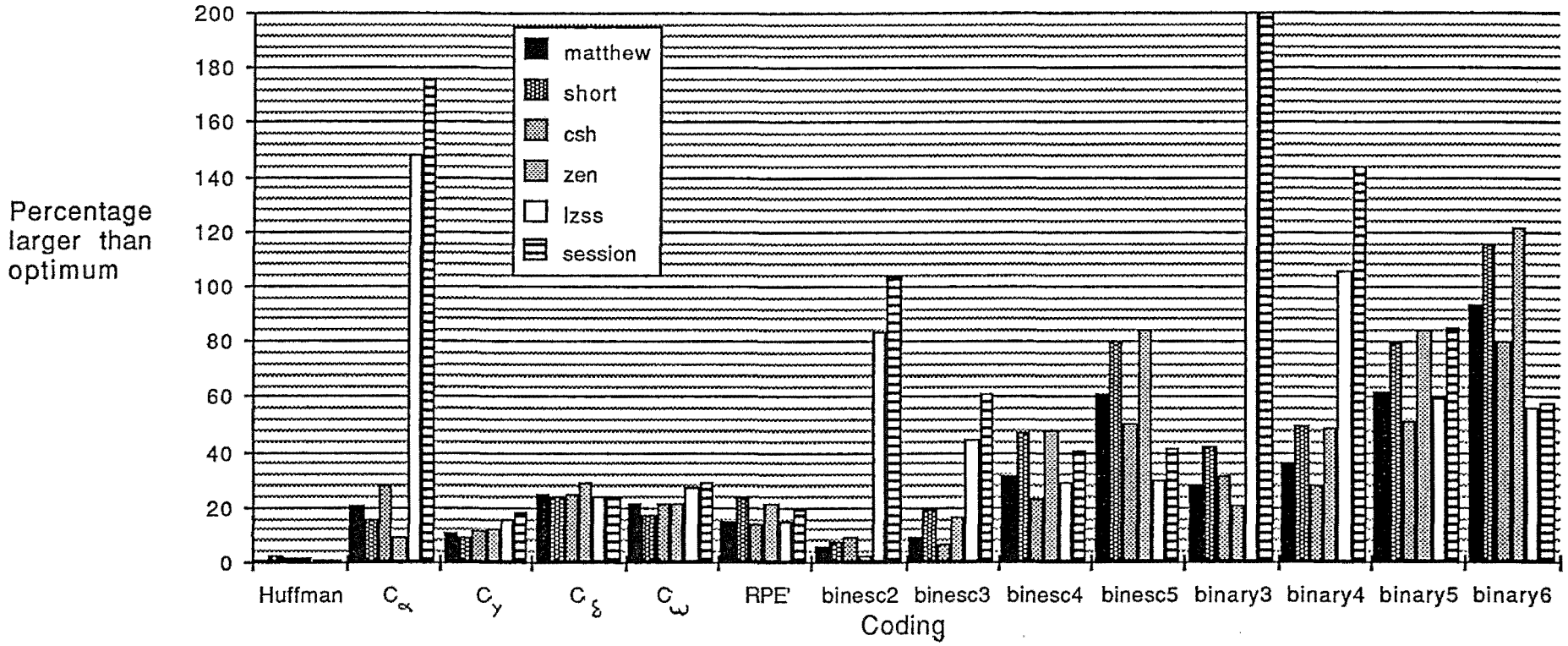
As well as the codes in appendix C, the codes binary3, binary4, binary5 and binary6 were evaluated. These codes simulate the number of bits used by covers when they are limited to 3, 4, 5 and 6 bits respectively, as in LZSS.

The best possible coding using an integer number of bits is achieved by Huffman coding, and this was also evaluated. Finally, the optimum coding possible is evaluated, which is the entropy of the covers i.e.

$$\Sigma\, P(i)\, \log_2 P(i), \qquad \text{where } P(i) = N(i) / \Sigma_j\, N(j)$$

Figure 6.2 compares each coding method, showing how much more space each method uses above the optimum as a percentage. As was already known from experiments with LZSS, binary3 and binary4 perform well with the English texts (matthew, short, csh and zen), while binary5 and binary6 perform well with the non-English texts (lzss and session). The fixed code which performed best all round was C_γ, and in fact was not much further from the optimum than Huffman coding. C_γ always performed better than the binary codes used by LZSS.

Figure 6.2: Performance of the codings in Appendix C for representing the cover in an LZSS pointer.



A new version of LZSS, using C_γ for pointer covers, was implemented, and called $LZSS_\gamma$. Experiments were performed with a window of 8192 characters. For this size window, the smallest pointer is 15 bits (1 for a flag, 13 for the reach and 1 for the smallest cover), so the usual LZSS algorithm would code two characters using a pointer (since to code them as explicit characters would use 16 bits). In practice it was found that what is gained by coding character pairs as pointers does not offset the extra bits needed to code larger phrases using pointers, and so it turns out that slightly better compression is achieved if pointers are used to code phrases of *more than* two characters. The CRs of this scheme and LZSS are compared in Table 6.2, showing a small improvement of 1 or 2 percent for most of the benchmark texts. Fortunately the C_γ coding is very simple, so this small improvement is achieved at very little cost. Also, the choice of the parameter, F,which is critical for LZSS to perform well, is trivial for $LZSS_\gamma$.

text	LZSS	$LZSS_\gamma$
matthew	43.25	42.05
short	54.72	52.86
csh	40.80	39.79
zen	48.33	47.70
lzss	25.64	24.10
session	25.04	23.65

Table 6.2: CRs (%) for LZSS and $LZSS_\gamma$ coding of benchmark texts

6.3.2 Pointer reaches

In the symbol-wise equivalent of LZSS, the reaches were coded as a series of probabilities, ending with an invalid character. Encoding this invalid character for each phrase corresponds to a pointer choosing a particular phrase from the window when there are other phrases(s) which would have given the same match length. If there are no other

phrases then $c(a_i a_{i+1} \ldots a_{i+x}) = 1$, and $p(\chi)$ is coded using log 1 (= 0) bits. This small redundancy allows LZ77 decoders to be very fast and simple. Eliminating it corresponds to keeping lists of the *unique* phrases in the sample, for lengths of 1 to F, as done by LZJ. The LZSS scheme reduces the effect of this problem by not allowing short phrases, and therefore reducing the chances of such repeated phrases. Maintaining such a list would make the decoder considerably more complex, and it is unlikely that much saving would be achieved.

The reaches of pointers make up the largest part of the compressed file (Figure 6.1), and so reducing their size will have a significant effect on the CR. The following experiment was performed to investigate better methods for coding the reach of a pointer.

The size of reaches (distance from coding position to position of phrase in window) was recorded for the benchmark files, using $LZSS_\gamma$ with a window of 8192 characters. The coding of the first 8192 characters was ignored, since the size of reaches is restricted there. The entropy of the reaches recorded is given in Table 6.3. No results were obtained for the text "short" because it contains less than 8192 characters.

text	Entropy of reach	Huffman
matthew	12.65	12.69
short	-	-
csh	12.02	12.03
zen	11.53	11.54
lzss	8.43	8.51
session	10.34	10.40

Table 6.3: Coding of reaches (average bits per reach) for benchmark files.

The largest value that the entropy could have had in Table 6.3 is 13 bits per reach, which occurs when the reach sizes are evenly distributed (non-skew). The entropies for the English texts are close to this value, indicating that phrases are almost as likely to be taken from any place in the window as another, and thus the use of a simple 13 bit binary code is near optimal. The non-English texts showed a more skew distribution. All of the variable length codes of Appendix C were applied to the distributions, but only Huffman's code achieved any improvement over the simple binary code. Although a small saving could be made by applying Huffman coding to the reaches, this will not be adopted here because of the complexity it adds to the scheme.

There is a useful improvement that can be made to the reaches, revealed by the symbolwise equivalent. The sample used to calculate probabilities in the symbol-wise equivalent is the N characters before the encoding position. This seems to be a reasonable choice, except when coding the first N characters of a text. Instead of assuming that the N characters preceding the text are blanks, it would make more sense to start the sample size at zero, and increase it to N during the coding of the first N characters. For LZ77 and LZSS, this corresponds to using less bits to encode the target of a pointer when encoding the first N characters. In general, to encode a pointer target when the encoding position is character number i, the minimum of $\lceil \log_2 i \rceil$ and $\lceil \log_2 N \rceil$ bits should be used. This is similar to the approach used by LZ78.

The growing sample was incorporated with $LZSS_\gamma$ to form the scheme $LZSS_{\gamma S}$. Recall that the decision at each coding step of whether to use a pointer or a character is based on the minimum length of a pointer. Practical experiments with $LZSS_{\gamma S}$ showed that a pointer should be at least 3 bits shorter than the characters it encodes, to make it worthwhile. This is a purely empirical result, but consistent for all of the benchmark files. The same phenomenon was observed for $LZSS_\gamma$ i.e. that the smaller space used to code large phrases more than offsets the space lost by coding small phrases as explicit characters.

text	LZSS	$LZSS_{\gamma}$	$LZSS_{\gamma S}$
matthew	43.25	42.05	41.93
short	54.72	52.86	49.14
csh	40.80	39.79	39.51
zen	48.33	47.70	47.06
lzss	25.64	24.10	23.25
session	25.04	23.65	23.46

Table 6.4: CRs (%) for LZSS, $LZSS_{\gamma}$ and $LZSS_{\gamma S}$ coding of benchmark texts

$LZSS_{\gamma S}$ is compared with $LZSS_{\gamma}$ in Table 6.4. The results show that $LZSS_{\gamma S}$ performs a little better than $LZSS_{\gamma}$ particularly for smaller files. As the value of N chosen for LZSS becomes larger than the size of the file being encoded, the compression ratio increases. For $LZSS_{\gamma S}$, choosing values of N larger than the file size has no effect on the CR, so no compression is lost by choosing a value of N which is too large. The criterion for choosing N is now to make it as large as is possible given the amount of memory available. This is equivalent to the symbol-wise scheme making the sample as large as possible.

6.3.3 Characters

Rather than coding every explicit character in 7 bits, a variable length code could be applied to these characters to achieve some space saving. Table 6.5 shows the entropy (zero- and first-order) of the explicit characters in the $LZSS_{\gamma S}$ coded benchmark files, and the average number of bits per explicit character achieved by applying some existing compression schemes to these characters.

text	0-order	1-order	compact	Macwrite	$LZSS_{\gamma S}$	diff63
matthew	5.5	4.8	5.6	8.2	6.5	6.3
short	5.3	3.5	6.0	7.2	7.0	6.8
csh	5.8	4.5	6.0	8.3	6.7	6.7
zen	5.3	4.3	5.6	7.5	6.6	6.3
lzss	5.6	3.5	6.3	8.0	7.1	6.9
session	5.8	4.3	6.1	8.9	6.4	6.6

Table 6.5: Average number of bits per character achieved by coding the explicit characters in an $LZSS_{\gamma S}$ compressed file.

The results show that the probability distribution of these explicit characters has a higher entropy than characters in the original texts, and so they are harder to compress. $LZSS_{\gamma S}$ already codes characters in 7 bits, so the simplest scheme, Macwrite, offers no saving. The other schemes were considered too complex to incorporate with $LZSS_{\gamma S}$, considering the savings achieved.

The *diff63* scheme was constructed specifically for the problem at hand. Experiments showed that the first 63 distinct characters in a file contain almost all the characters ever used. This is exploited by diff63 using an adaptive two-level scheme. Initially characters are coded in 7 bits, until 63 different characters have been encountered. From there on, those 63 characters are coded in 6 bits, and all other characters are coded as a 6 bit escape followed by an explicit 7 bit character. Only a small saving is achieved by diff63, but it is simple to implement.

Table 6.6 shows the CRs achieved by incorporating diff63 with $LZSS_{\gamma S}$, to form $LZSS_{\gamma Sd}$. Although a small improvement is achieved, it barely justifies the extra implementation effort and memory required.

text	$LZSS_{YS}$	$LZSS_{YSd}$
matthew	41.93	41.34
short	49.14	48.60
csh	39.51	39.25
zen	47.06	46.44
lzss	23.25	23.14
session	23.46	23.06

Table 6.6: CRs (%) for $LZSS_{YSd}$ coding of benchmark texts

6.3.4 Flags

In order to find out if any saving could be made on the 1-bit flags, the entropy of the flags generated by $LZSS_{YSd}$ was measured for the benchmark files. The entropies were measured adaptively, using up to 4th-order contexts. The results in Table 6.7 show that using 0-order prediction, flags can be coded in less than 0.5 bits on average. A small improvement was achieved by using first-order prediction. Examination of the Markov transition probabilities showed that this was mainly because a pointer is more likely to be followed by another pointer than a character. For the files short, lzss, and session, it was also observed that a character is more likely to be followed by another character.

text	size of context				
	0	1	2	3	4
matthew	0.41	0.37	0.37	0.37	0.37
short	0.48	0.44	0.45	0.45	0.45
csh	0.43	0.39	0.38	0.38	0.38
zen	0.42	0.39	0.39	0.39	0.39
lzss	0.48	0.43	0.42	0.42	0.42
session	0.49	0.41	0.40	0.39	0.39

Table 6.7: Entropy (bits per flag) of flags in a compressed $LZSS_{\gamma Sd}$ file, using immediately preceding flags for the context.

Because flags only use one bit, the only way they can use less space, apart from arithmetic coding, is to use a form of blocking, such as run length coding. If blocking is used, several flags would be transmitted together in one block, in advance. Although this is feasible, the saving would be negligible because flags make up such a small proportion (less than 5%) of a compressed file, and it would be at the expense of less speed and more implementation effort.

6.4 The LZB scheme

Small savings have been made by improving the coding of the cover and reach of pointers, and of characters. Some of these improvements were originally exposed by the symbol-wise equivalent given at the beginning of the chapter. Although the savings were small, the new codes for covers and reaches make the choice of parameters for LZSS less critical, and add little to the complexity of the scheme, or compression time. The scheme which offered the most benefits for the minimum effort was $LZSS_{\gamma S}$, which codes covers using Elias' C_γ and codes reaches using an increasing number of bits until the window is full. $LZSS_{\gamma S}$ is labelled LZB in Table 2.2, and is described in full in appendix B. To verify

that LZB will give better compression in practice than the competing LZ schemes, LZC and LZSS, the three schemes were used to compress a large number of files which are common on UNIX systems. The files are divided into three classes:

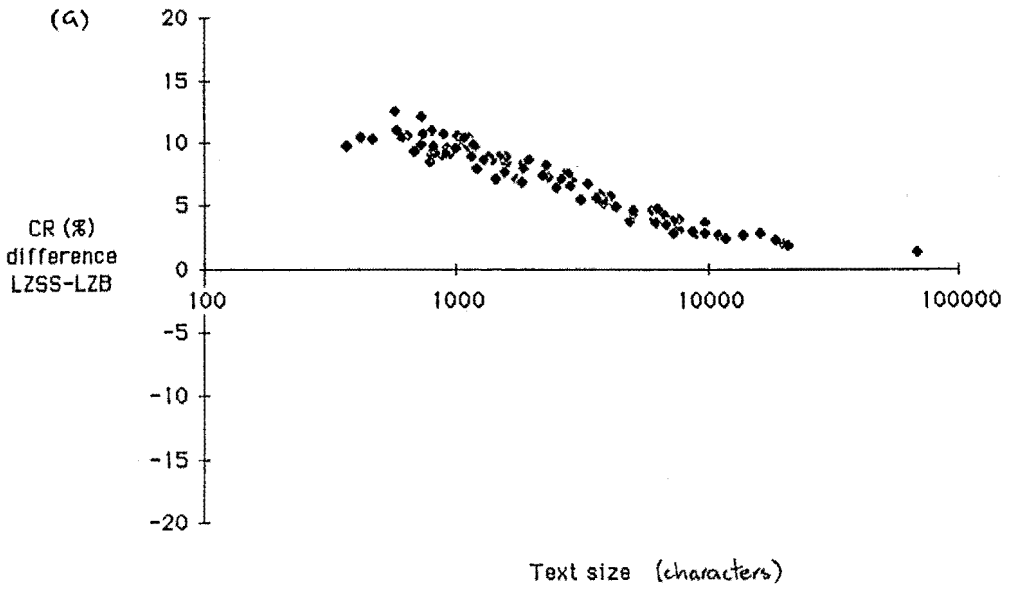
(a) Manual files: 317 formatted help files from the directory /usr/man/cat1,

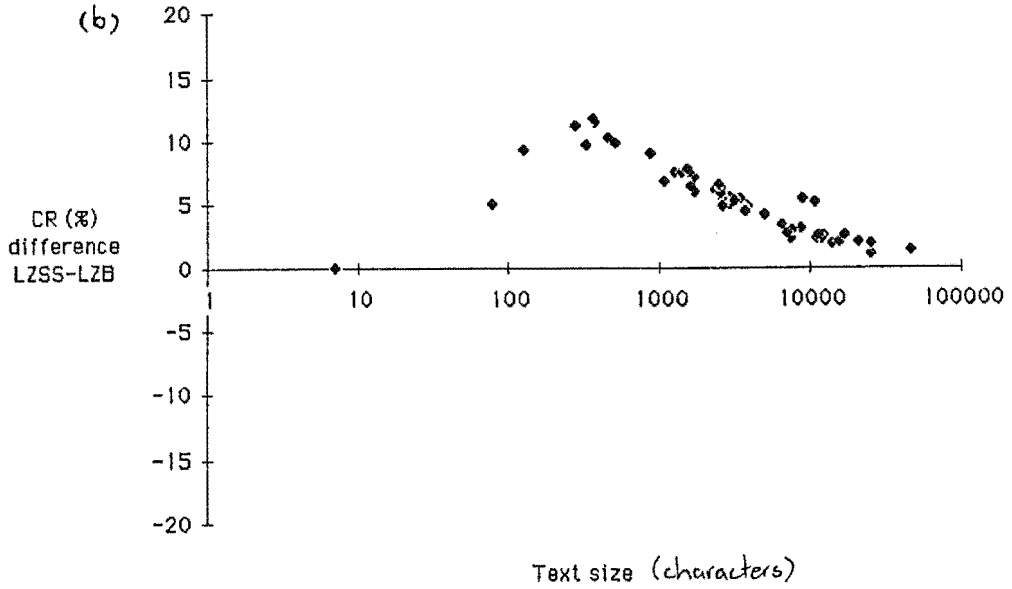
(b) Source files: the source code (mainly in the C programming language) of 53 common UNIX commands, from the directory /usr/src/bin,

(c) System files: 32 files from the directory /etc, mainly containing administrative records.

Figure 6.3 shows the difference between the CR achieved by LZSS and LZB, for each file. A positive difference indicates that LZB gave better compression. Figure 6.4 shows the corresponding difference for LZC. In both cases, LZB gave better compression than LZC and LZSS for every file, with the exception of LZC for two files which contained 7 characters. For these 7-character files, the output of LZB was 12 bytes while LZC stored 11 bytes. The files expanded because both compression schemes have a small overhead for identifying a compressed file, and storing the compression parameters. Often the compression achieved by LZB was significantly better than LZC and LZSS. Figure 6.5 shows the actual compression achieved by LZB. The only time the compression ratio is greater than 100% is for some very small files. Also, for the manual and source files, the CR correlates closely with the size of the file, indicating that LZB is consistent in the amount of compression it achieves for a given type of text. The system files contained a wider variety of text than the manual and source files, and so the CR is less predictable. LZB is particularly well suited to the applications where files are encoded once and decoded many times, because LZB decoding is very fast and requires as little as 8 kbytes of memory.

A feature of LZB (and LZSS) which may be useful in some situations is that if the flag bit used to signal a character is chosen to be the same as the default for the most significant





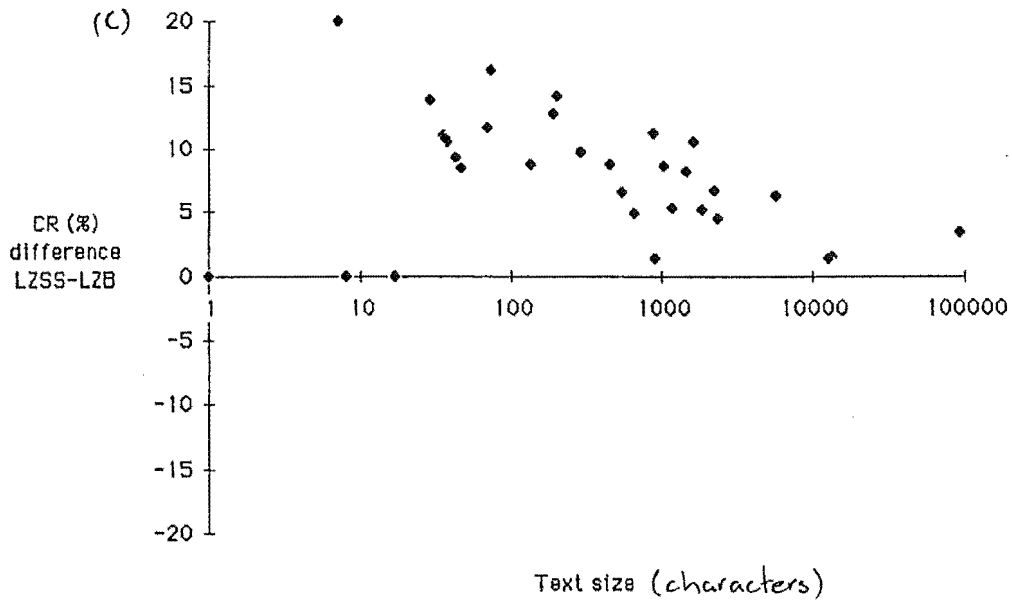
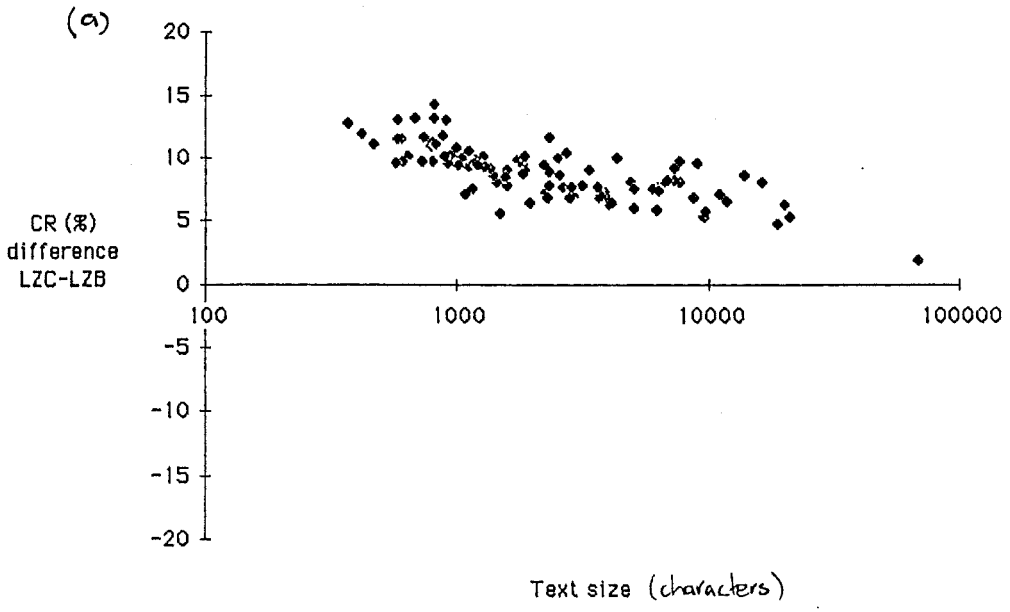
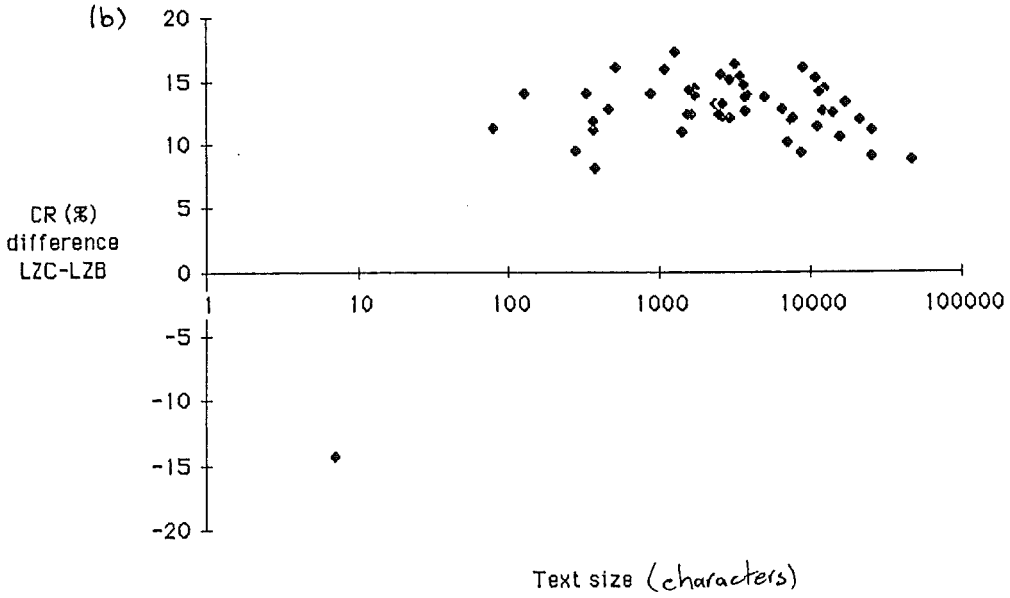


Figure 6.3: Difference between CR for LZSS and LZB coding of:
(a) manual files
(b) source files
(c) system files





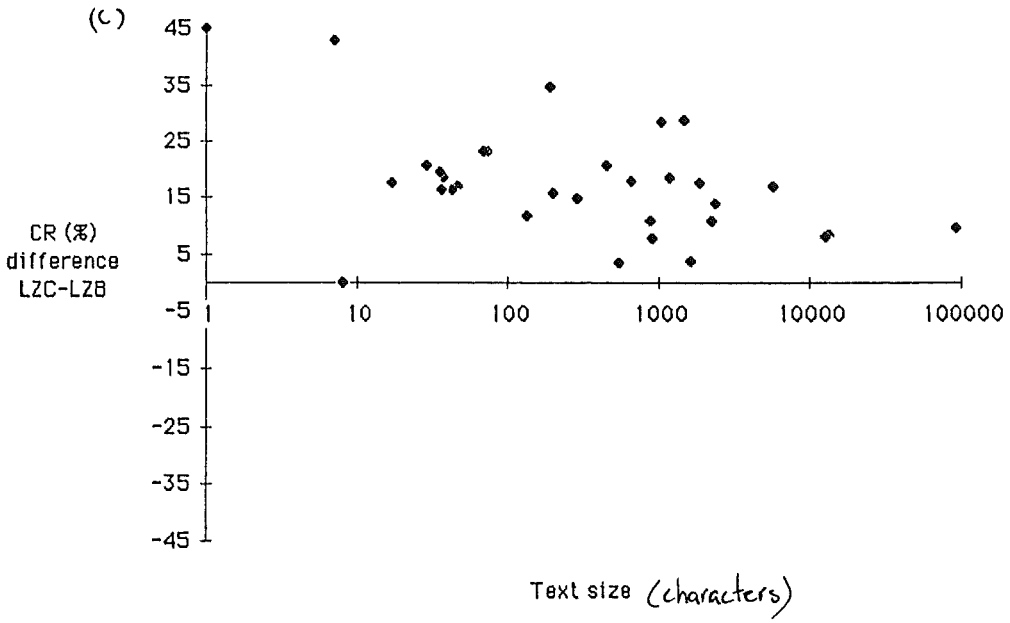
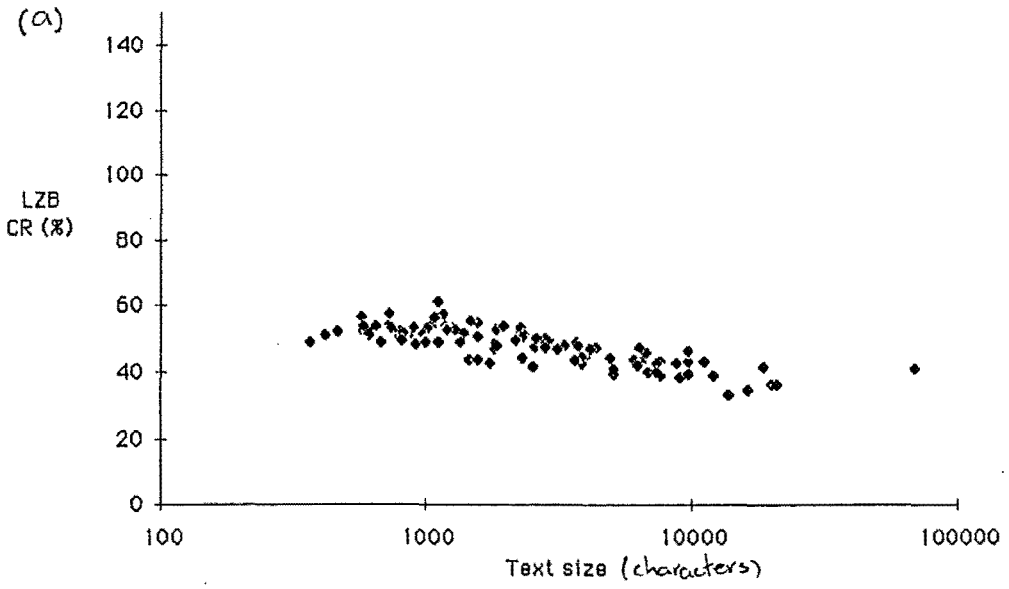
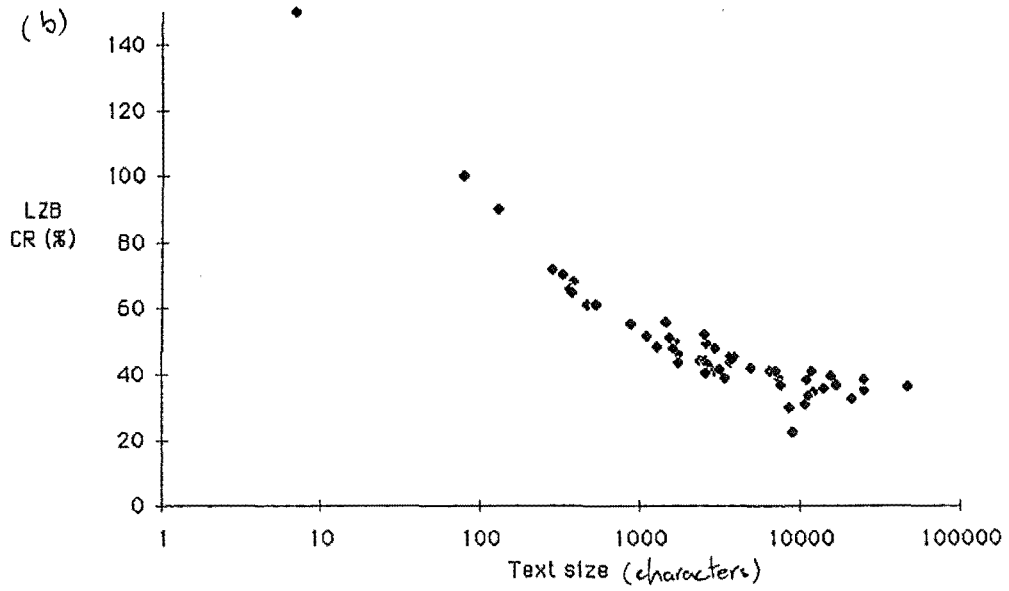


Figure 6.4:
Difference between CR for LZC and LZB coding of:
(a) manual files
(b) source files
(c) system files





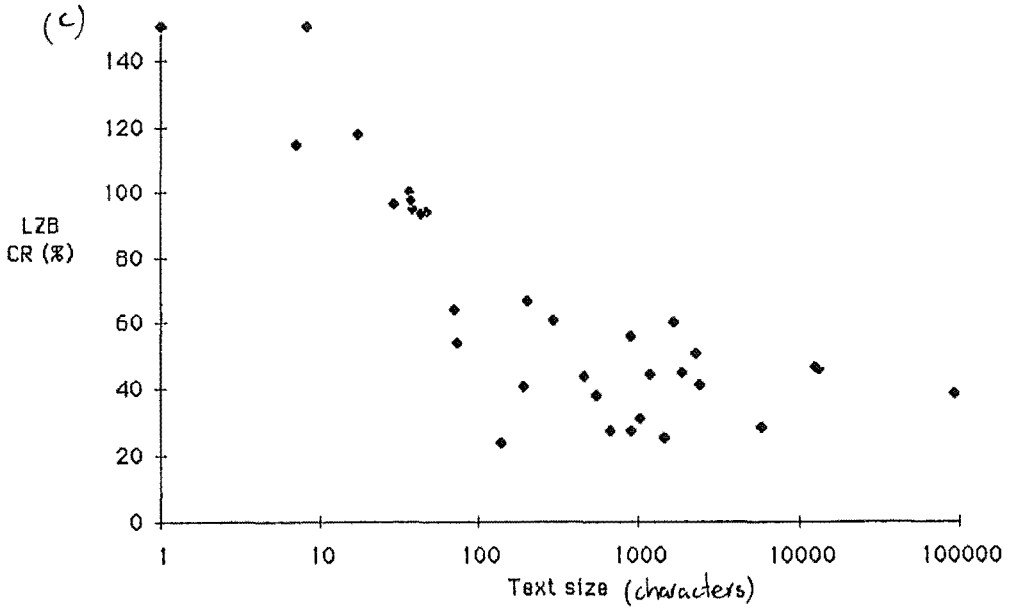


Figure 6.5:
CR for LZB coding of:
(a) manual files
(b) source files
(c) system files

bit (MSB) of characters in an uncompressed file (assuming no parity bit is stored), then decoding an unencoded file will output the file unchanged. This may be useful when reading a mixed group of encoded and unencoded files, since there is no need to treat them differently. If the parameters N and F are to be included in the encoded file, they should be stored in bytes with the MSB set to the opposite of the default.

Chapter 7
Faster Ziv-Lempel coding

A problem common to the LZ77, LZSS and LZB schemes is that, although the encoding speed is O(n) for a text of size n, it can be slow in practice because of the need to search the window for a longest match for the lookahead buffer. Rodeh, Pratt and Even address this problem by applying McCreight's data structure [Rodeh 81], but this approach is unnecessarily complicated. VLSI implementations of LZ schemes are given by Gonzalez-Smith and Storer [Gonzalez 85]. The implementations applicable to LZ77, LZSS and LZB are very fast, although they use O(N) processors.

There is still a need for a fast encoding algorithm which can run on a conventional computer without specialised hardware. The time consuming step of encoding is a search for a longest match for the lookahead buffer. Section 7.1 defines this *longest match* problem formally, and section 7.2 shows that a binary search tree data structure can be used to achieve faster encoding. Details of implementation of the binary search tree algorithm are given in section 7.3, and the performance of the algorithm is evaluated in section 7.4.

7.1 The longest match problem

The following notation is used. If s is a string of n characters, $s = s_1 s_2 \ldots s_n$, then s_i may be written as s(i), and the substring $s_i \ldots s_j$ may be written as s(i,j).

Given two strings s and t, of length n, the *match* between the strings, M(s,t) is defined to be the length of the longest common prefix of s and t.

Given a set of q strings of length n, $U = \{ u_1, u_2 \ldots u_q \}$, the *longest match* , LM(s,U), is a function on a string s and the set of strings, U, which returns the ordered pair

$(i,M(s,u_i))$, where $M(s,u_i)$ is maximal over all the strings in U. The ordered pair gives the position and size of a longest match for s in U. This need not be unique, but any one of the different possible values may be used, since the main concern is the *length* of the match.

To code the r^{th} character of a string s , a window w, of N characters, and a lookahead buffer l, of F characters is used:

$$w = s(r + F - N, r + F - 1)$$

$$l = s(r , r + F - 1)$$

To be consistent with the implementation of the window described for LZSS (see 2.6.6), a special case is made for numbering the string w. Each character in w is given the same index that it has in the original string, modulo N. This means that s(j) corresponds to w(j mod N), provided that s(j) is in the window. Substrings are extracted from the window using wraparound, so if j>k then w(j,k) is evaluated as if a copy of w is concatenated to the end of w.

The strings of length F in the window are denoted by

$$x_j = w(j , j + F - 1) \quad , \; j = r+F-N \ldots r \; .$$

Note that $l = x_r$.

X is the set of all distinct strings of length F in the window, apart from l i.e.

$$X = \{ \, x_j : j = r+F-N \ldots r-1 \, \} \; .$$

Where there are two substrings $x_i = x_j : i \neq j$, the substring which entered the window most recently is chosen for inclusion in X.

To perform LZ77, LZSS or LZB coding, a procedure is needed to evaluate

$$(g,h) = LM(l,X).$$

The first h elements of the lookahead buffer may then be coded with the pointer (g,h).

For example, if the string s = "abcabcbacbababcabc ..." is being encoded with the parameters N = 11 and F = 4, and coding is up to character 12, then the window is as shown in Figure 7.1.

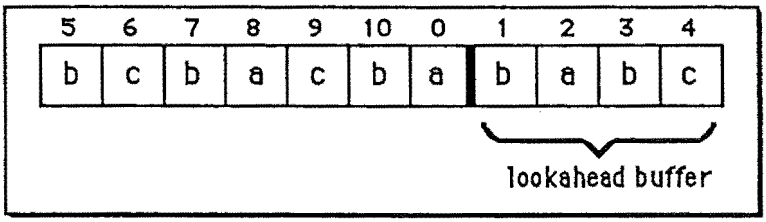


Figure 7.1: An LZ coder

and $l = x_1 = babc$, and

$x_5 = bcba$, $x_6 = cbac$, $x_7 = bacb$, $x_8 = acba$,

$x_9 = cbab$, $x_{10} = baba$, $x_0 = abab$.

By inspection, the longest match is x_{10} i.e. LM(l,x) = (10,3).

A straightforward algorithm would evaluate $M(l,x_i)$ for each member of X. This requires up to N-F evaluations of M to obtain each pointer in the coded form of the input string, and runs very slowly.

7.2 Binary tree algorithm

Suppose the set $X \cup \{ l \}$ (i.e. all strings of length F in the window) is sorted into lexicographical order, and l (alias x_r) is found adjacent to x_a and x_b in the sorted list, and $x_a \leq l \leq x_b$. Then either LM(l,X) = (a, M(l,x_a))

or LM(l,X) = (b, M(l,x_b))

i.e. a longest match for l in w is found at the beginning of either x_a or x_b.

For example, if the set X from the previous example is sorted, the list becomes:

x_0	x_8	x_{10}	l	x_7	x_5	x_9	x_6
abab	acba	baba	babc	bacb	bcba	cbab	cbac

and a longest match for l should be found at the beginning of x_{10} or x_7.

The following observations show how an ordered binary search tree can be used to efficiently yield the two substrings lexicographically adjacent to the lookahead buffer `l`, and hence the two candidates for a longest match. See Wirth [Wirth 76] for a definition of binary search trees, and for algorithms to insert and delete nodes in a tree. Symmetric order in the tree corresponds to the lexicographical order of the substrings, that is, for any node x_i, all nodes in its left subtree are lexicographically less than x_i, and all nodes in its right subtree are lexicographically greater than x_i. The nodes in the tree for this application contain the substrings in the set X. In practice, the node x_i need only store i, since x_i can be obtained from w (i.e. $x_i = w(i,i+F-1)$). It will be shown later that the tree can be stored compactly using an array of N nodes.

The special case where x_i is to be inserted in the tree, but $x_j = x_i$ is already in the tree is ignored in the following discussion. Finding a longest match for x_i in this case is trivial because the insertion algorithm will come across x_j, and x_j can be used as the longest match for x_i (it is impossible to have a longer match). x_j is then replaced with x_i. The less trivial case, where there is no exact match in the tree, is now considered.

Suppose $X = \{ x_i : i = r+F-N \ldots r-1 \}$ has been inserted into an empty binary search tree according to the usual rules. When `l` is inserted in the tree, observe that both x_a and x_b will be on the path from the root to where the insertion is made.

This can be proved by considering the situation after `l` has been inserted in the tree. If x_a is not on the path to `l`, then `l` and x_a must have at least one common ancestor. If x_p is the most recent common ancestor then it must have x_a in its left subtree, and `l` in its right subtree, which implies that $x_a < x_p <$ `l` . This is a contradiction because x_a and `l` are adjacent, so x_a must be on the path to `l` . A similar argument shows that x_b must be on the path to `l`.

For example, Figure 7.2 shows that if the set X of the previous examples is inserted into a binary search tree, followed by `l`, in the order $x_5, x_6, \ldots x_{10}, x_0,$ `l` , then x_{10} and x_7 appear on the path to `l`.

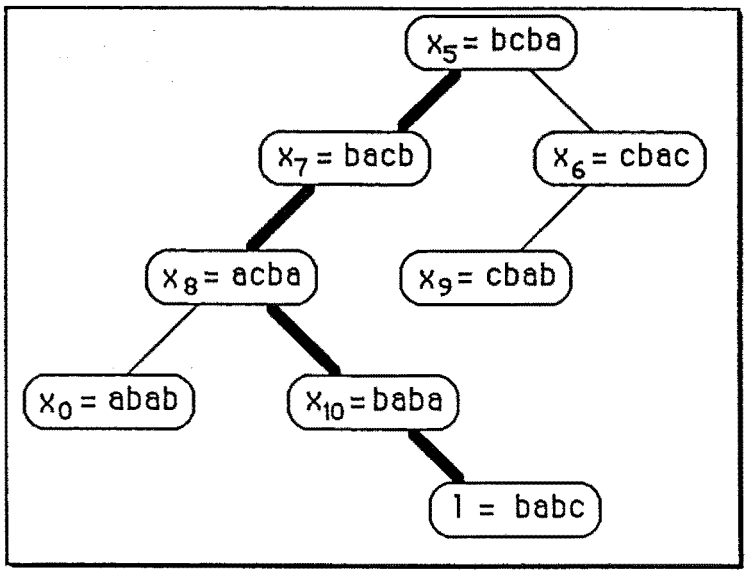


Figure 7.2: Using a binary tree to find the longest match

In fact, it is easily shown that either x_a or x_b will be the parent of 1, and the other candidate for the longest match is found using the rules:

(1) If x_a is the parent of 1 then x_b is the node where the insertion path last turned left.

(2) If x_b is the parent of 1 then x_a is the node where the insertion path last turned right.

7.3 Implementation of the tree algorithm

During encoding, the tree is continuously updated as the window changes. Each time the window moves along a character, one character, s(i), leaves the window. The tree is searched for the associated x_i in the tree, and if it is found, it is deleted. Also, one character, s(j), enters the window, and a new lookahead buffer, x_{j-F+1}, is inserted in the tree. Whenever a prefix of the lookahead buffer is to be coded as a pointer, the LM function is used.

The LM function is evaluated as a by-product of inserting the new lookahead buffer in the tree. During the insertion, the match length is recorded for the most recent left and right turns (Figure 7.3). The new parent of the lookahead buffer is one candidate for the longest

match, and depending on whether the lookahead buffer was inserted as a left or right son, the node where the insertion path last turns right or left respectively is the other candidate.

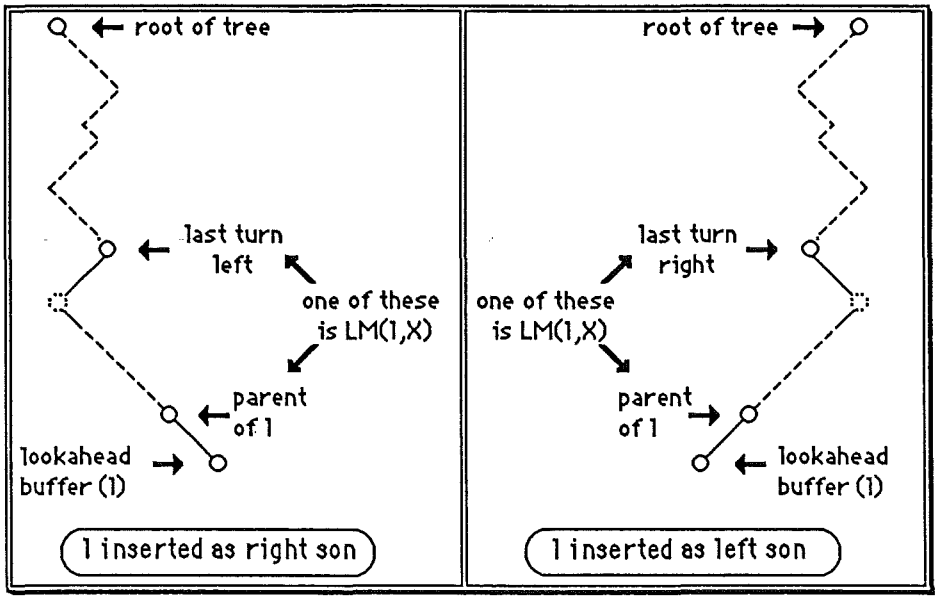


Figure 7.3: Implementation of the binary tree algorithm

The candidate with the longer match length is the longest match for `l`. Thus the longest match length is evaluated with only *one* comparison of match lengths, although the normal character comparisons for inserting `l` in the tree will have been made. Using this algorithm with a binary search tree, the coding time for each character will be the time to perform a tree insertion and deletion.

7.3.1 Details of the tree data structure

To simplify garbage collection, the nodes for the tree are drawn from an array of N nodes, where the i^{th} node in the array is used if x_i is to be stored in the tree. It has already been pointed out that a node need only store an index to the string it represents, rather than the entire string. Using the array of nodes means that there is no need to store even the index to the string associated with x_i, because the index to the string in the window is the same as the index to the node in the tree.

Each node contains two pointers to any sons the node has. The null pointer is represented by a pointer to the node itself. The pointers are actually indexes to the array of N nodes.

Deletion in a binary tree is traditionally performed by searching for the node to be deleted, and then adjusting the tree around it. Because x_i can be located directly at element *i* of the tree array, there is no need to search the tree. However, because deletion requires access to the parent of the node to be deleted, each node must store a pointer to its parent, as well as to its two sons. A deleted node is marked by the parent pointer pointing to the node itself.

The data structure resulting from this is two arrays :

(1) the window of N characters, and

(2) an array of N nodes, with each node containing three indexes (two sons, one parent) of other nodes.

7.3.2 Initialising the tree

For LZ77 and LZSS the initial window must be inserted in the tree before the coding may be started. If the window is initialised with N-F blanks and the first F characters of the text, there are only F distinct substrings to insert in the tree. The order in which these substrings are inserted in the tree is important. With the ASCII character set, inserting the substrings form left to right causes the initial binary tree to degenerate to a linked list which is traversed frequently in subsequent insertions. A simple solution is to insert the first F substrings from right to left. This still generates a linked list, but the list is only traversed to insert substrings beginning with blanks. The most appropriate method for initialising the tree will depend on the initial window and the lexical ordering of the character set.

7.4 Performance of the tree algorithm

7.4.1 Analysis of running time

If there are M characters in the input string, there will be M insertions and deletions performed on the tree. Insertion and deletion times depend only on the constants N and F, and on the "randomness" of the text, so running time for the tree algorithm is O(M).

Each insertion requires a probe into the tree of O(log N) string comparisons for a reasonably balanced tree, but O(N) in the worst case. If worst case behaviour must be avoided, a balanced tree, such as an AVL tree [Wirth 76], could be used. On average, each string comparison requires only a couple of character comparisons, but in the worst case F character comparisons could be required. The time used for deletion is independent of N and F. Note that the tree algorithm performs a tree insert and delete for each *input character*, while the straightforward algorithm performs a search for each *output character or pointer.*

7.4.2 Empirical results for running time

LZSS encoding times were measured for four of the benchmark texts using the straight-forward algorithm and the tree algorithm.The algorithms were programmed in the C language on a VAX 11/750. The results for different values of N are shown in Figure 7.4. As expected, the straight-forward algorithm encoding time increases linearly with N, and the tree algorithm time increases (approximately) logarithmically with N. The tree algorithm was faster than the straightforward algorithm for all files when N > 512. The encoding speeds (characters per second) for "matthew" have improved from 18 to 319 for N=8192, and from 52 to 383 for N=2048.

Encoding the files with better CRs was slower because they used larger lookahead buffers and because regularities in those files created less balanced trees.

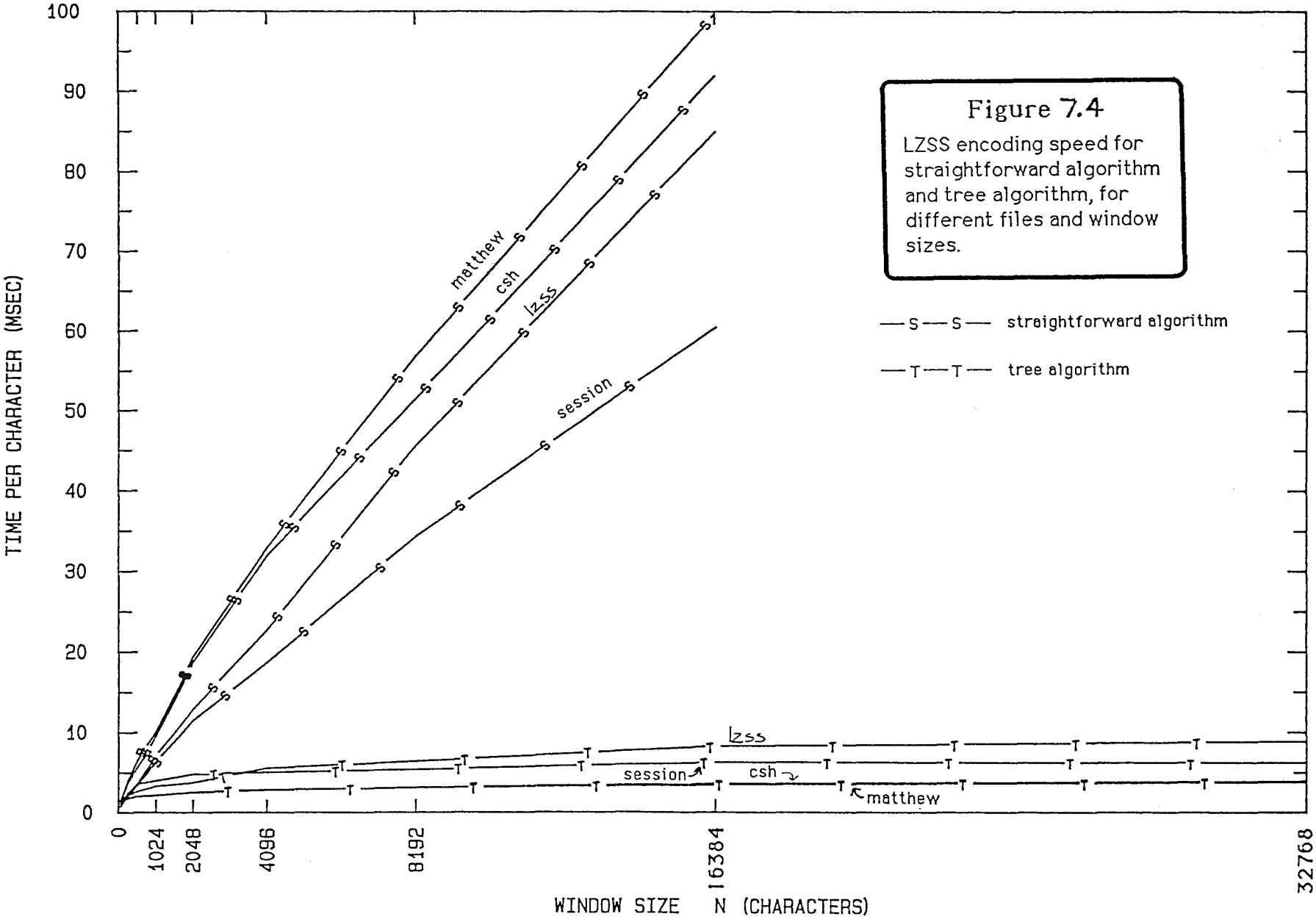


Figure 7.4

LZSS encoding speed for straightforward algorithm and tree algorithm, for different files and window sizes.

7.4.3 Memory usage

Assuming 8 bits for each character in the window, the total memory required for the window (N characters) and tree (array of N nodes) is

$$8N + 3N\lceil \log_2 N \rceil \text{ bits}.$$

For N = 8192, this is 47 kbytes (or 56 kbytes if the pointer size is made a multiple of 8 bits). By comparison, two copies of McCreight's tree [McCreight 76] (as used by Rodeh, Pratt and Even [Rodeh 81]) require

$$2(4N \log_2 N + 3N.8 + 4N)$$

$$= 56N + 8N \log_2 N \text{ bits},$$

or 160 kbytes for N = 8192.

Chapter 8
Conclusion

A representative sample of text compression schemes has been presented. The most important result given in this thesis is that each of these text compression schemes disguises some form of variable-order Markov modelling. In a variable-order Markov model (VOMM) each symbol of a text is predicted by some finite number of immediately preceding symbols, which are the *Markov context*. The size of this context may vary, but if it is too large the prediction becomes unreliable due to a lack of samples. Different approaches to text compression are distinguished from each other by their different methods of choosing a suitable context size. It can be argued that a VOMM is an inadequate model for text because such models cannot represent the grammatical structures present in most texts. The conclusion is that more powerful models will have to be found if better compression is to be achieved. These models will have to be appropriate for the type of text being compressed.

Usually a scheme was shown here to be equivalent to a VOMM by giving a symbol-wise equivalent for the scheme. This symbol-wise equivalent assigns a probability to (predicts) each symbol encoded. Seeing how probabilities are assigned has given insights into how each scheme achieves compression. In particular, the size of the context used for prediction in different symbol-wise equivalents is consistent with the ranking of schemes by empirical experiments in chapter 2. For example, the Ziv-Lempel schemes have been shown to use a gradually growing Markov context to predict symbols. The context increases by one symbol each time an input symbol is encoded and is reset to zero symbols when it becomes too long. By contrast, the PPM and DMC compression schemes always use the longest context for which a sample is available. The use of longer contexts should give better prediction, and correspondingly better compression. The results of the empirical

comparisons of the schemes confirm this, as PPM and DMC gave significantly better compression than any Ziv-Lempel scheme.

It is impossible to show exhaustively that all TC schemes use a VOMM because of the large and growing number of schemes described in the literature, but two useful tools have been provided for working with new schemes. The first is the decomposition of Greedy Macro (GM) schemes (chapter 5), which shows that every GM scheme is no more powerful than a VOMM. A GM scheme uses a set of strings, M, and a coding function C(m), which is associated with each string $m \in M$. The input text is parsed for strings in M using a greedy algorithm, and the strings are replaced with their associated code. To decompose a GM scheme to a VOMM it is necessary only to identify the set M, and the length of each code $|C(m)|$. The second tool provided is the Finite Context Automaton (FCA), which has been shown also to implement a form of VOMM, and was used to show that the DMC compression scheme uses a VOMM.

The discovery of symbol-wise equivalents has not been only of theoretical interest. In chapter 6 a symbol-wise equivalent of the LZ77 scheme revealed inefficiencies in the underlying model, and an improved scheme LZB was constructed by the author. In experiments on over 400 texts, LZB consistently gave better compression than competing LZ schemes. Although LZB encoding is slow, decoding is very fast, and requires very little memory (typically 8 kbytes). LZB is therefore very suitable for applications where a text is to be encoded once and decoded many times, or when the text can be encoded on a large computer but must be decoded on a small machine. Examples are: on-line help files and news, decentralised databases [Urrows 84], teletext [Money 79], and electronic books [Weyer 85].

Some results of investigation of models more powerful than VOMMs have already appeared in the literature. Ozeki [Ozeki 74a, 74b, 75] has investigated the use of a Context Free Grammar (CFG) as a model. His compression scheme requires that a CFG is known for the language of the text being encoded and so it has a limited practical application. However, he does show that parsing a string using a CFG model results in a sequence of

productions which exhibit Markovian behaviour. A good understanding of Markov modelling will therefore be needed as more sophisticated models are developed.

In conclusion, the work presented in this thesis has achieved an empirical *and* theoretical comparison of many different types of text compression schemes in the literature, showing where each stands in the trade off between compression, speed, memory requirements, and complexity. By removing the ad hoc approaches that have been disguising Markov models, the path to better compression is now clearer, and the better understanding of Markov models will be valuable in the investigation of more sophisticated models.

Appendix A
Better OPM/L Text Compression

The following paper is [Bell 86], appearing in:

IEEE Trans. Commun., December 1986

Abstract

An OPM/L data compression scheme suggested by Ziv and Lempel, LZ77, is applied to text compression. A slightly modified version suggested by Storer and Szymanski, LZSS, is found to achieve compression ratios as good as most existing schemes for a wide range of texts. LZSS decoding is very fast, and comparatively little memory is required for encoding and decoding.

Although the time complexity of LZ77 and LZSS encoding is O(M) for a text of M characters, straightforward implementations are very slow. The time consuming step of these algorithms is a search for the longest string match. Here a binary search tree is used to find the longest string match, and experiments show that this results in a dramatic increase in encoding speed. The binary tree algorithm can be used to speed up other OPM/L schemes, and other applications where a longest string match is required. Although the LZSS scheme imposes a limit on the length of a match, the binary tree algorithm will work without any limit.

1 Introduction

Many advantages can be obtained from compressing text that is stored on or transmitted by a computer, and many schemes to perform the compression have been described in the literature ([5],[14],[15],[16],[18]). One difficulty encountered when designing a Text Compression (TC) scheme is to trade off the reduction in size of the text, the encoding and decoding speed, the encoding and decoding memory required, and the ease of implementation. Earlier schemes in the literature tended to use a small amount of memory and CPU time [15], but recently both of these have become cheaper, and later schemes have concentrated on achieving the best possible compression [1].

Here we extend a scheme proposed by Ziv and Lempel [22] which will be referred to as LZ77. Many of the results can be applied to the OPM/L class of TC schemes [16], of which LZ77 is a member. An OPM/L (original pointer macro restricted to left pointers) scheme replaces a substring in a text with a pointer to a previous (left) occurrence of the substring in the text. The pointer represents the position and size of the substring in the original text. These restrictions make fast single pass decoding straightforward. Single pass

encoding is also usually possible, which is important if the compressed text is to be transmitted with bounded delay as it is compressed.

Compression schemes have been evaluated here on textual data such as documents, source code, on line manuals, textual databases and data transmitted to terminals . However, the coding schemes are easily applied to general binary files.

The LZ77 scheme restricts the reach of the pointer to approximately the previous N characters, effectively creating a 'window' of N characters which are used as a sliding dictionary. Pointers are chosen using a 'greedy' algorithm, which permits single pass encoding. LZ77 is described in more detail in section 2.

The use of a window has several advantages:

(1) the amount of memory required for encoding and decoding is bounded by the size of the window, and is typically no more that 8 kbytes,

(2) for many types of text, and for sufficiently large N, the window is a good dictionary for the substring which follows, because it will usually contain the same language, style and topic, and

(3) all pointers can have fixed size fields.

Ziv and Lempel were more concerned with the theoretical properties of LZ77 than with its practical application. They have since described another scheme, LZ78 [23], for which encoding is significantly faster and easier to implement [6] than LZ77. However, it does not use a window and restricts the substrings which can be targets of pointers. Consequently, encoding and decoding memory are unbounded, and decoding is slower and as complicated as encoding. A variation of LZ78 suitable for hardware implementation (LZW) is described by Welch [18].

Arithmetic codes [7] have also appeared since LZ77 with many of the same advantages and disadvantages as LZ78. Langdon [6] shows that LZ78 is equivalent to an arithmetic code.

The main areas in which LZ77 does not outperform LZ78 and arithmetic codes are the compression achieved, and the encoding speed. These two aspects of LZ77 will be improved in sections 3 and 5 respectively.

Section 2 describes the LZ77 scheme, and in section 3 an improvement of LZ77 suggested by Storer and Szymanski [16], LZSS, is described. LZSS has many desirable properties, except that like LZ77, the encoding speed is very slow.

Section 4 formally describes the time consuming step of LZ77 and LZSS encoding, and the main contribution of this paper is an improved algorithm to perform that step, in section 5. Experimental results are given in section 6 which show the improved algorithm to be in excess of ten times faster than a straightforward implementation of the schemes.

In what follows, the amount of compression achieved in experiments is measured by the Compression Ratio (CR), which is defined as the size of the compressed file expressed

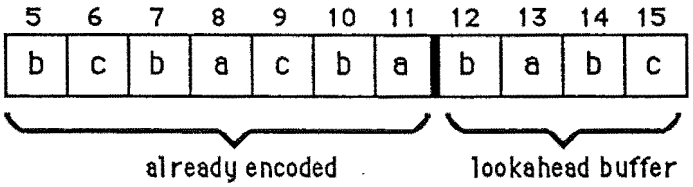
as a percentage of the original file [5]. The size of the original file will be calculated assuming 8 bits for each character.

2 The LZ77 scheme

An LZ77 encoder is parameterised by N, the size of the 'window' on the text, and F, the maximum length of a substring that may be replaced by a pointer. The window is used as a 'sliding dictionary', and substrings are chosen from the window using a greedy algorithm.

Encoding of the input string proceeds from left to right. At each step of the encoding, a section of the input text is available in a window of N characters. Of these, the first N-F characters have already been encoded and the last F characters are the 'lookahead buffer'.

For example, if the string s = abcabcbacbababcabc ... is being encoded with the parameters N = 11 and F = 4, and character 12 is to be encoded next, the window is:



Initially the first N-F characters of the window are (arbitrarily) blanks, and the first F characters of the text are loaded into the lookahead buffer.

The already encoded part of the window is searched to find the longest match for the lookahead buffer. The match may overlap with the lookahead buffer, but obviously cannot be the lookahead buffer itself. In the example, the longest match for the "babc" is "bab", which starts at character 10.

The longest match is then coded into a triple <i,j,a>, where i is the offset of the longest match from the lookahead buffer, j is the length of the match, and a is the first character which did not match the substring in the window. In the example, the output triple would be <2,3,'c'>. The window is then shifted right j+1 characters, ready for another coding step.

A window of moderate size, typically $N \leq 8192$, can work well for a variety of texts because:

(1) Common words and fragments of words occur regularly enough in a text to appear more than once in a window. For example, in English "the","of","pre-","-ing"; source program keywords "while","if","then".

(2) Specialist words tend to occur in clusters. For example, a paragraph on a technical topic, or local identifiers in a procedure of a source program.

(3) Less common words may be made up of fragments of common words

(4) Runs of characters are coded compactly. For example, k blanks may be coded recursively as <?,?,' '> <1,k-1,?>.

The amount of memory required for encoding and decoding is limited to the size of the window. The offset (i) in a triple can be represented in $\lceil \log_2(N-F) \rceil$ bits, and the number of characters (j) covered by the triple in $\lceil \log_2 F \rceil$ bits. The time taken at each step is bounded to N-F substring comparisons, which is constant, so the time used for encoding is O(n) for a text of size n.

Decoding is very simple and fast. The decoder maintains a window in the same way as the encoder but, instead of searching for a match in the window, it copies the match from the window using the triple given by the encoder.

The main disadvantage of LZ77 is that, although the encoding step requires O(1) time, a straightforward implementation can require up to (N-F)*F character comparisons, typically in the order of several thousands. LZ77 is therefore best for the situation where a file is to be encoded once (preferably on a fast computer) and decoded many times, possibly on a small machine. Examples of these situations are on-line help files and manuals, de-centralised data bases [17], teletext [12], and electronic books [19].

3 The LZSS scheme

The output of the LZ77 scheme is a series of triples, which can also be viewed as a series of strictly alternating pointers and characters. In what follows, the triple <i,j,a> will now be denoted by the pointer (i,j), and the character a. In Storer and Szymanski's work on OPM/L schemes [16], they suggest that the Lempel-Ziv algorithms use a free mixture of pointers and characters; a character being used only when a pointer would take more space than the characters it codes. Here the LZ77 scheme has Storer and Szymanski's suggestion incorporated to produce what will be called the LZSS algorithm. Suppose a pointer uses the space of p unencoded characters. The LZSS algorithm is:

```
while lookahead buffer not empty do
        get a pointer (offset,length) to the longest match in the
             window for the lookahead buffer
        if length > p then
             output the pointer (offset,length)
             shift window length characters
        else
             output first character in lookahead buffer
             shift window one character
```

An extra bit is added to each pointer or character to distinguish between them. The output is packed so that there are no unused bits.

Implementations of the LZSS and LZ77 encoders and decoders can be simplified by numbering the input text characters modulo N. The window is an array of N characters. To shift in character number r (modulo N), it is simply necessary to overwrite element r of the array, which implicitly shifts out character r-N (modulo N). Instead of an offset, the first

element of an (i,j) pointer can be a position in the array (0 .. N-1). This means that i is capable of indexing substrings which start in the lookahead buffer. These F unused values of i cause negligible deterioration in the compression ratio provided that F << N, and they can be used for special messages such as 'end of file'.

The first element of a pointer can be coded in $\lceil \log_2 N \rceil$ bits. Because the second element of the pointer can never have any of the values 0,1, ... ,p , it can be coded in $\lceil \log_2 (F - p) \rceil$ bits. Including the pointer flag bit, a pointer requires

$$1 + \lceil \log_2 N \rceil + \lceil \log_2 (F - p) \rceil \text{ bits.}$$

Input characters will be assumed to be ASCII, although the results are easily adapted for EBCDIC and other codes. An output character requires one flag bit plus 7 bits to represent an ASCII character, a total of 8 bits.

The coding parameters N and F can be specified in terms of the number of bits they are to be coded in, which will be denoted as n and f respectively. N and F are calculated from :

$$p = \lfloor (1 + n + f)/8 \rfloor$$

$$N = 2^n$$

$$F = 2^f + p$$

The performance of LZSS depends on the values chosen for the coding parameters, N and F. The CR and speed of LZSS were measured in practical experiments, for various values of N and F, using five different text files. Different types of text were chosen to give an indication of the performance of LZSS in different situations. The files used were :

(1) matthew	a book from the Good News Bible	(139,521 characters)
(2) short	the first 100 lines of matthew	(4,510 characters)
(3) csh	an online manual on a UNIX system	(60,997 characters)
(4) lzss	a commented C program to code files using the LZSS scheme	(25,750 characters)
(5) session	a transcript of text transmitted to a terminal during an edit-compile-run session	(57,127 characters)

Only csh contained tabs, and new lines were encoded as a single character. The first three files contained nroff formatting commands.

Initially the Compression Ratio (CR) was measured for each file, for all combinations of window sizes with $6 \le n \le 15$ ($64 \le N \le 32768$), and lookahead buffers with $1 \le f \le 7$ ($3 \le F \le 130$). The best CRs were usually obtained with f=4 or f=5 ($17 \le F \le 34$), and the CR did not vary greatly for different values of f around this range. The best CR achieved for each value of N is shown in Fig 1, as a function of N.

The results show that the CR decreases uniformly as N is increased, except for the files 'short' and 'lzss' when the window was larger than the text. Having a window the same size as the text means that pointers are able to reference any previous substring in the text, and increasing the size of the window further will not make new substrings available.

Consequently, the CR cannot be improved by enlarging the window, and it actually gets worse because the size of a pointer increases with N.

It seems from figure 1 that the optimum CR is achieved when the window is about the same size as the text. In practice, the value of N may be limited by the encoding time and memory available for the window. Figure 1 shows that the CR will suffer very little with the constraint $N \leq 8192$.

Figure 1 also shows how the CRs vary for different files. The best CRs were achieved with the files containing limited vocabularies (the C program and the terminal session), while the worst CRs were obtained with the short file, which contained too little information for the encoder to build a good model of the language.

In table 1, the CRs for LZSS (N = 2048 and N = 8192) are compared with those for other Text Compression schemes. For the LZSS and LZ77 schemes, the value of the parameter F was chosen for the best CR. That value is shown in the table. The schemes compared with LZSS are:

(1) LZ77 N = 8192

(2) LZ78

(3) LZW

(4) Arithmetic coding. Because of the diversity of models used by arithmetic schemes, one scheme (used as a comparison by Langdon and Rissanen [8]) has been chosen as a representative. The scheme uses a first-order Markov model to encode each character. The text file is first scanned to determine all first-order probabilities, and then coded using them.

Coding schemes described in the literature use various models. Some use lower order Markov contexts [8] , and consequently yield worse CRs, while others use higher order contexts [1], which demand large amounts of memory, but achieve better CRs.

(5) An adaptive Huffman code. The scheme given by Gallager [3] has been used, although Cormak and Horspool [2] have since discovered an improved adaptive Huffman scheme.

Compression Ratios (%)

	LZSS N=8192	LZSS N=2048	LZ77 N=8192	LZ78	LZW	Arith-metic	Adaptive Huffman
matthew	43.3 F=10	49.2 F=18	49.7 F=15	50.4	47.9	42.6	58.7
short	54.7 F=10	52.8 F=18	65.4 F=15	65.5	64.7	40.5	59.5
csh	40.8 F=18	46.5 F=18	47.5 F=15	52.4	49.7	43.4	59.7
lzst	25.6 F=34	28.1 F=34	30.7 F=63	38.6	33.8	27.4	44.3
session	25.0 F=66	27.3 F=34	28.4 F=63	39.6	44.3	36.6	62.7

Table 1

Table 1 shows that LZSS performs very well when compared with the other schemes. The only scheme likely to perform better than LZSS is a higher-order Markov model arithmetic scheme, such as the one described by Cleary and Witten [1].

Other indices of performance

Comparative figures are given in Table 2 for other important performance indices - the speed and memory requirements for encoding and decoding. Memory requirements were calculated from the size of the main data structures of each scheme, and do not include the program code. Speed measurements were obtained by encoding and decoding the file "matthew" using programs written in the C language on a VAX 11/750. The LZ77 and LZSS implementations used a straightforward linear search to find the longest string matches. The memory requirements are independent of the text, except for LZ78, which was measured for the file "matthew".

		LZSS N=8192	LZSS N=2048	LZ77 N=8192	LZ78	LZW	Arith-metic	Adapt. Huff.
Speed (characters per second)	Encode	18	52	24	5300	5700	-	990
	Decode	13,600	10,900	15,200	10,060	8400	-	1300
Memory (kbytes)	Encode	8	2	8	350	48	32 to 1400	8
	Decode	8	2	8	135	12	32 to 1400	8

Table 2

Table 2 shows that LZSS performs very well in all areas except encoding speed, which is atrociously bad. As with LZ77, LZSS is a good candidate when a text is to be encoded once and decoded many times, and is likely to give better compression ratios.

4 The longest match problem

LZ77 is a successor of an earlier algorithm, LZ76 [9]. LZ76 obtains an indication of the 'complexity' of a text, rather than compressing it. However, because this complexity measure was made by looking for substrings which have occurred previously in the text, LZ76 is easily adapted for use as a TC scheme, which was done by Rodeh, Pratt and Even [13]. It is very similar to LZ77, but does not use a window i.e. pointers have an unbounded reach. A straightforward implementation of LZ76 might require $O(n^2)$ time for a text of n characters, because to encode each substring, all the text to the left of the substring must be searched for the longest match. Rodeh, Pratt and Even reduced this to O(n) by employing a data structure discovered by McCreight [11]. They then adapted the LZ76 data structure for LZ77.

McCreight's data structure is basically a complex multiway tree. Rather than delete nodes from the tree as characters leave the LZ77 window, Rodeh, Pratt and Even use several trees, and a whole tree is thrown away once all the text it represents is out of the window. Although an improvement is undoubtedly achieved, the data structure and algorithm are much more complicated than is needed for LZ77, because it was approached as an adaptation of LZ76.

VLSI implementations of OPM/L schemes are given by Gonzalez-Smith and Storer [4]. The implementations applicable to LZ77 and LZSS are very fast, although they use O(N) processors.

There is still a need for a fast LZSS encoding algorithm which can run on a conventional computer without specialised hardware. Section 5 shows that a binary search tree data structure can be used to achieve faster encoding. As a preliminary, the longest match step is defined formally.

The following notation is used. If s is a string of n characters, $s = s_1 s_2 \ldots s_n$, then s_i may be written as s(i), and the substring $s_i \ldots s_j$ may be written as s(i,j).

Given two strings s and t, of length n, the *match* between the strings, M(s,t) is defined to be the length of the longest common prefix of s and t.

Given a set of q strings of length n, $U = \{ u_1, u_2 \ldots u_q \}$, the *longest match* , LM(s,U), is a function on a string s and the set of strings, U, which returns the ordered pair (i , $M(s,u_i)$), where $M(s,u_i)$ is maximal over all the strings in U. The ordered pair gives the position and size of a longest match for s in U. This need not be unique, but any one of the different possible values may be used, since the main concern is the *length* of the match.

Problem definition

To code the r^{th} character of a string s under LZSS or LZ77, a window, w, of N characters, and a lookahead buffer, l, of F characters is used:

$$w = s(r + F - N, r + F - 1)$$

$$l = s(r, r + F - 1)$$

To be consistent with the implementation of the window in section 3, a special case is made for numbering the string w. Each character in w is given the same index that it has in the original string, modulo N. This means that s(j) corresponds to w(j mod N), provided that s(j) is in the window. Substrings are extracted from the window using wraparound, so if j>k then w(j,k) is evaluated as if a copy of w is concatenated to the end of w.

The strings of length F in the window are denoted by

$$x_j = w(j, j + F - 1) \quad , \; j = r+F-N \ldots r \; .$$

Note that $l = x_r$.

X is the set of all distinct strings of length F in the window, apart from l i.e.

$$X = \{ x_j : j = r+F-N \ldots r-1 \} .$$

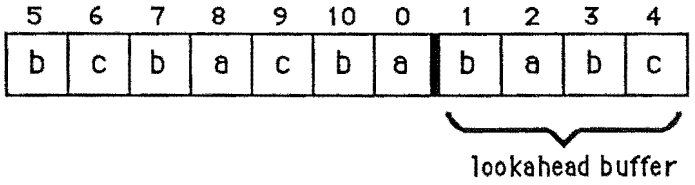
Where there are two substrings $x_i = x_j : i \neq j$, the substring which entered the window most recently is chosen for inclusion in X.

To perform LZ77 or LZSS coding, a procedure is needed to evaluate

$$(g,h) = LM(l,X).$$

The first h elements of the lookahead buffer may then be coded with the pointer (g,h).

For example, if the string s = abcabcbacbababcabc ... is being encoded with the parameters N = 11 and F = 4, and coding is up to character 12, then the window is:



and $l = x_1$ = babc, and

x_5 = bcba, x_6 = cbac, x_7 = bacb, x_8 = acba,

x_9 = cbab, x_{10} = baba, x_0 = abab.

By inspection, the longest match is x_{10} i.e. LM(l,x) = (10,3).

The straightforward algorithm evaluates M(l,x_i) for each member of X. This requires up to N-F evaluations of M to obtain each pointer in the coded form of the input string, s. The figures in table 2 have shown that this algorithm runs very slowly.

5 Binary tree algorithm

Suppose the set X ∪ { l } (i.e. all strings of length F in the window) is sorted into lexicographical order, and l (alias x_r) is found adjacent to x_a and x_b in the sorted list, and $x_a \leq l \leq x_b$. Then either LM(l,X) = (a, M(l,x_a))

$$\text{or} \qquad LM(\, l,X) = (\, b, M(\, l,x_b)\,)$$

i.e. a longest match for l in w is found at the beginning of either x_a or x_b.

For example, if the set X from the previous example is sorted, the list becomes:

x_0	x_8	x_{10}	l	x_7	x_5	x_9	x_6
abab	acba	baba	babc	bacb	bcba	cbab	cbac

and a longest match for l should be found at the beginning of x_{10} or x_7.

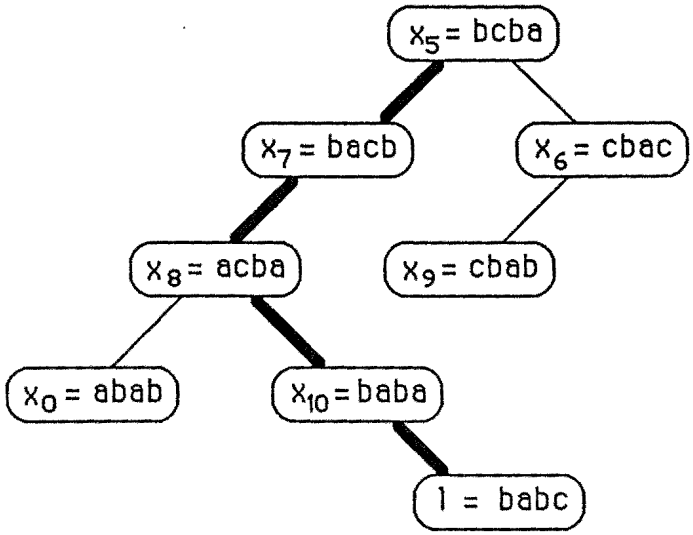
The following observations show how an ordered binary search tree can be used to efficiently yield the two substrings lexicographically adjacent to the lookahead buffer, l, and hence the two candidates for a longest match. See Wirth [20] for a definition of binary search trees, and for algorithms to insert and delete nodes in a tree. Symmetric order in the tree corresponds to the lexicographical order of the substrings, that is, for any node x_i, all nodes in its left subtree are lexicographically less than x_i, and all nodes in its right subtree are lexicographically greater than x_i. The nodes in the tree for this application contain the substrings in the set X. In practice, the node x_i need only store i, since x_i can be obtained from w (i.e. x_i = w(i,i+F-1)). It will be shown later that the tree can be stored compactly using an array of N nodes.

The special case where x_i is to be inserted in the tree, but $x_j = x_i$ is already in the tree is ignored in the following discussion. Finding a longest match for x_i in this case is trivial because the insertion algorithm will come across x_j, and x_j can be used as the longest match for x_i (it is impossible to have a longer match). x_j is then replaced with x_i. The less trivial case, where there is no exact match in the tree, is now considered.

Suppose X = { x_i : i = r+F-N ... r-1 } has been inserted into an empty binary search tree according to the usual rules. When l is inserted in the tree, observe that both x_a and x_b will be on the path from the root to where the insertion is made.

This can be proved by considering the situation after l has been inserted in the tree. If x_a is not on the path to l, then l and x_a must have at least one common ancestor. If x_p is the most recent common ancestor then it must have x_a in its left subtree, and l in its right subtree, which implies that $x_a < x_p < l$. This is a contradiction because x_a and l are adjacent, so x_a must be on the path to l . A similar argument shows that x_b must be on the path to l.

For example, if the set X of the previous examples is inserted into a binary search tree, followed by 1, in the order $x_5, x_6, \ldots x_{10}, x_0$, 1 , then x_{10} and x_7 appear on the path to 1.



In fact, it is easily shown that either x_a or x_b will be the parent of 1, and the other candidate for the longest match is found using the rules:

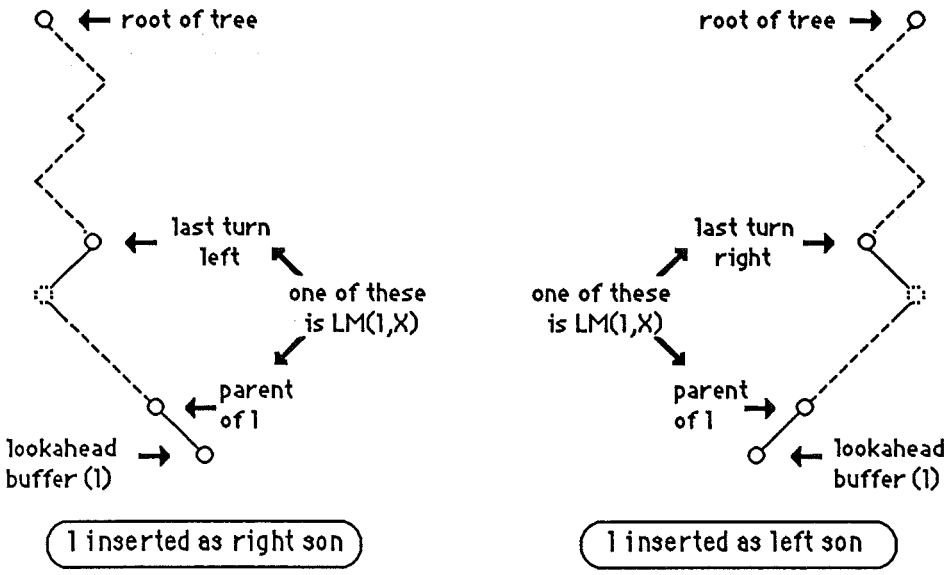
(1) If x_a is the parent of 1 then x_b is the node where the insertion path last turned left.

(2) If x_b is the parent of 1 then x_a is the node where the insertion path last turned right.

Implementation of LZSS using the tree algorithm

During LZSS encoding, the tree is continuously updated as the window changes. Each time the window moves along a character, one character, s(i), leaves the window. The tree is searched for the associated x_i in the tree, and if it is found, it is deleted. Also, one character, s(j), enters the window, and a new lookahead buffer, x_{j-F+1}, is inserted in the tree. Whenever a prefix of the lookahead buffer is to be coded as a pointer, the LM function is used.

The LM function is evaluated as a by-product of inserting the new lookahead buffer in the tree. During the insertion, the match length is recorded for the most recent left and right turns. The new parent of the lookahead buffer is one candidate for the longest match, and depending on whether the lookahead buffer was inserted as a left or right son, the node where the insertion path last turns right or left respectively is the other candidate.



The candidate with the longer match length is the longest match for `l`. Thus the longest match length is evaluated with only *one* comparison of match lengths, although the normal character comparisons for inserting `l` in the tree will have been made. Using this algorithm with a binary search tree, the coding time for each character will be the time to perform a tree insertion and deletion.

Details of the tree data structure

To simplify garbage collection, the nodes for the tree are drawn from an array of N nodes, where the i^{th} node in the array is used if x_i is to be stored in the tree. It has already been pointed out that a node need only store an index to the string it represents, rather than the entire string. Using the array of nodes means that there is no need to store even the index to the string associated with x_i, because the index to the string in the window is the same as the index to the node in the tree.

Each node contains two pointers to any sons the node has. The null pointer is represented by a pointer to the node itself. The pointers are actually indexes to the array of N nodes.

Deletion in a binary tree is traditionally performed by searching for the node to be deleted, and then adjusting the tree around it. Because x_i can be located directly at element i of the tree array, there is no need to search the tree. However, because deletion requires access to the parent of the node to be deleted, each node must store a pointer to its parent, as well as to its two sons. A deleted node is marked by the parent pointer pointing to the node itself.

The data structure resulting from this is two arrays :

(1) the window of N characters, and

(2) an array of N nodes, with each node containing three indexes (two sons, one parent) of other nodes.

Initialising the tree

The initial window must be inserted in the tree before the coding may be started. If the window is initialised with N-F blanks and the first F characters of the text, there are only F distinct substrings to insert in the tree. The order in which these substrings are inserted in the tree is important. With the ASCII character set, inserting the substrings form left to right causes the initial binary tree to degenerate to a linked list which is traversed frequently in subsequent insertions. A simple solution is to insert the first F substrings from right to left. This still generates a linked list, but the list is only traversed to insert substrings beginning with blanks. The most appropriate method for initialising the tree will depend on the initial window and the lexical ordering of the character set.

Analysis of running time

If there are M characters in the input string, there will be M insertions and deletions performed on the tree. Insertion and deletion times depend only on the constants N and F, and on the 'randomness' of the text, so running time for the tree algorithm is O(M).

Each insertion requires a probe into the tree of O(log N) string comparisons for a reasonably balanced tree, but O(N) in the worst case. If worst case behaviour must be avoided, a balanced tree, such as an AVL tree [21], could be used. On average, each string comparison requires only a couple of character comparisons, but in the worst case F character comparisons could be required. The time used for deletion is independent of N and F. Note that the tree algorithm performs a tree insert and delete for each *input character*, while the straightforward algorithm performs a search for each *output character or pointer*.

6 Performance of the tree algorithm

LZSS encoding times were measured for four of the text files using the straight-forward algorithm and the tree algorithm. The results for different values of N are shown in figure 2. As expected, the straight-forward algorithm encoding time increases linearly with N, and the tree algorithm time increases (approximately) logarithmically with N. The tree algorithm was faster than the straightforward algorithm for all files when N > 512. The encoding speeds in table 2 (characters per second) have improved from 18 to 319 for N=8192, and from 52 to 383 for N=2048.

Encoding the files with better CRs was slower because they used larger lookahead buffers and because regularities in those files created less balanced trees.

Encoding memory

Assuming 8 bits for each character in the window, the total memory required for the window (N characters) and tree (array of N nodes) is

$$8N + 3N\lceil \log_2 N \rceil \text{ bits}.$$

For N = 8192, this is 47 kbytes (or 56 kbytes if the pointer size is made a multiple of 8 bits). By comparison, two copies of McCreight's tree [11] (as used by Rodeh, Pratt and Even [13]) require

$$2(4N \log_2 N + 3N.8 + 4N)$$

$$= \quad 56N + 8N \log_2 N \quad \text{bits},$$

or 160 kbytes for N = 8192.

7 Conclusion

A simple modification to the LZ77 scheme, as suggested by Storer and Szymanski [15], has produced a useful scheme, LZSS. Practical experiments have shown that LZSS performs at least as well as most existing TC schemes in all areas except encoding speed. We have used a binary search tree to bring encoding speed to within an order of magnitude of other schemes. LZSS is particularly suited to applications where a file is to be encoded once and decoded several times.

The binary tree algorithm presented is the most important contribution of this paper. It employs a well-known data structure to find the longest match for a string. It uses a moderate amount of memory to produce a large speedup from a simple linear search - the experiments have shown an increase in speed by a factor of 18 achieved with a memory increase from 8 kbytes to 56 kbytes.

The algorithm is easily generalized to any application where a longest string match is required for a slowly changing set of strings (for example, Mayne and James [9]). There need not be a limit on the maximum match length, and the text need not be limited to a window, so the binary tree approach is more powerful than a fixed-depth trie data structure. To search a string of n characters, O(n) memory is required, and for typical texts, the search time is O(log n).

Acknowledgements

The author would like to thank J.P. Penny, B.J. McKenzie, K. Pawlikowski and A.M. Moffat for their ideas and comments on this work.

References

[1] JG Cleary and IH Witten, Data compression using adaptive coding and partial string matching, IEEE Trans. Communications COM-32 n 4 pp 396 - 402 (1984)

[2] GV Cormack and RN Horspool, Algorithms for adaptive Huffman codes, Inf. Process. Lett., v 18 n 3 pp 159-165 (1984)

[3] RG Gallager, Variations on a theme by Huffman, IEEE Trans. Information Theory IT-24 n 6 pp 668 - 674 (1978)

[4] ME Gonzalez Smith and JA Storer, Parallel Algorithms for Data Compression, J. ACM, v 32 n 2 pp 344 - 373 (1985)

[5] Gotlieb, Hagerth, Lehot and Rabinowitz, A classification of compression methods and their usefulness for a large data processing centre, National Comp. Conference 44 pp 453 - 458 (1975)

[6] G Langdon, A note on the Ziv-Lempel model for compressing individual sequences, IEEE Trans. Information Theory IT-29 pp 284-287 (1983)

[7] G Langdon and J Rissanen, A simple general binary source code, IEEE Trans. Information Theory IT-28 pp 800 - 803 (1982)

[8] G Langdon and J Rissanen, A double-adaptive file compression algorithm, IEEE Trans. Communications COM-31 n 11 pp 1253-1255 (1983)

[9] A Lempel and J Ziv, On the complexity of finite sequences, IEEE Trans. Information Theory IT-22 n 1 pp 75-81 (1976)

[10] A Mayne and EB James, Information compression by factorising common strings, Computer Journal v 18 n 2 pp 157 - 160 (1975)

[11] EM McCreight, A space-economical suffix tree construction algorithm, J. ACM v 23 n 2 pp 262 - 272 (1976)

[12] SA Money, Teletext and Viewdata, Butterworth & Co., London (1979)

[13] M Rodeh, VR Pratt and S Even, Linear algorithm for data compression via string matching, J. ACM v 28 n 1 pp 16 - 24 (1981)

[14] DG Severance, A practitioner's guide to database compression, Inf. Syst. v 8 n 1 pp 51 - 62 (1983)

[15] M Snyderman and B Hunt, The myriad virtues of text compaction, Datamation, 1 Dec 1970 pp 36 - 40 (1970)

[16] JA Storer and TG Szymanski, Data compression via textual substitution, J. ACM v 29 n 4 pp 928 - 951 (1982)

[17] H Urrows and E Urrows, LaserData, Mnemos and other data disks : the race to store and retrieve with optics, Videodisc and Optical Disk, v 4 n 2 p 130 March - April (1984)

[18] TA Welch, A technique for high performance data compression, Computer v 17 n 6 pp 8 - 19 (1984)

[19] SA Weyer and AH Borning, A prototype electronic encyclopedia, ACM Trans. on Office Information Systems v 3 n 1 pp 63 - 88 (1985)

[20] N Wirth, Algorithms + Data Structures = Programs, Prentice-Hall inc., Englewood Cliffs, N.J., pp 201 - 211 (1976)

[21] ibid., pp 215-226

[22] J Ziv and A Lempel, A universal algorithm for sequential data compression, IEEE Trans. Information Theory IT-23 n 3 pp 337 - 343 (1977)

[23] J Ziv and A Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Information Theory IT-24 n 5 pp 530 - 536 (1978)

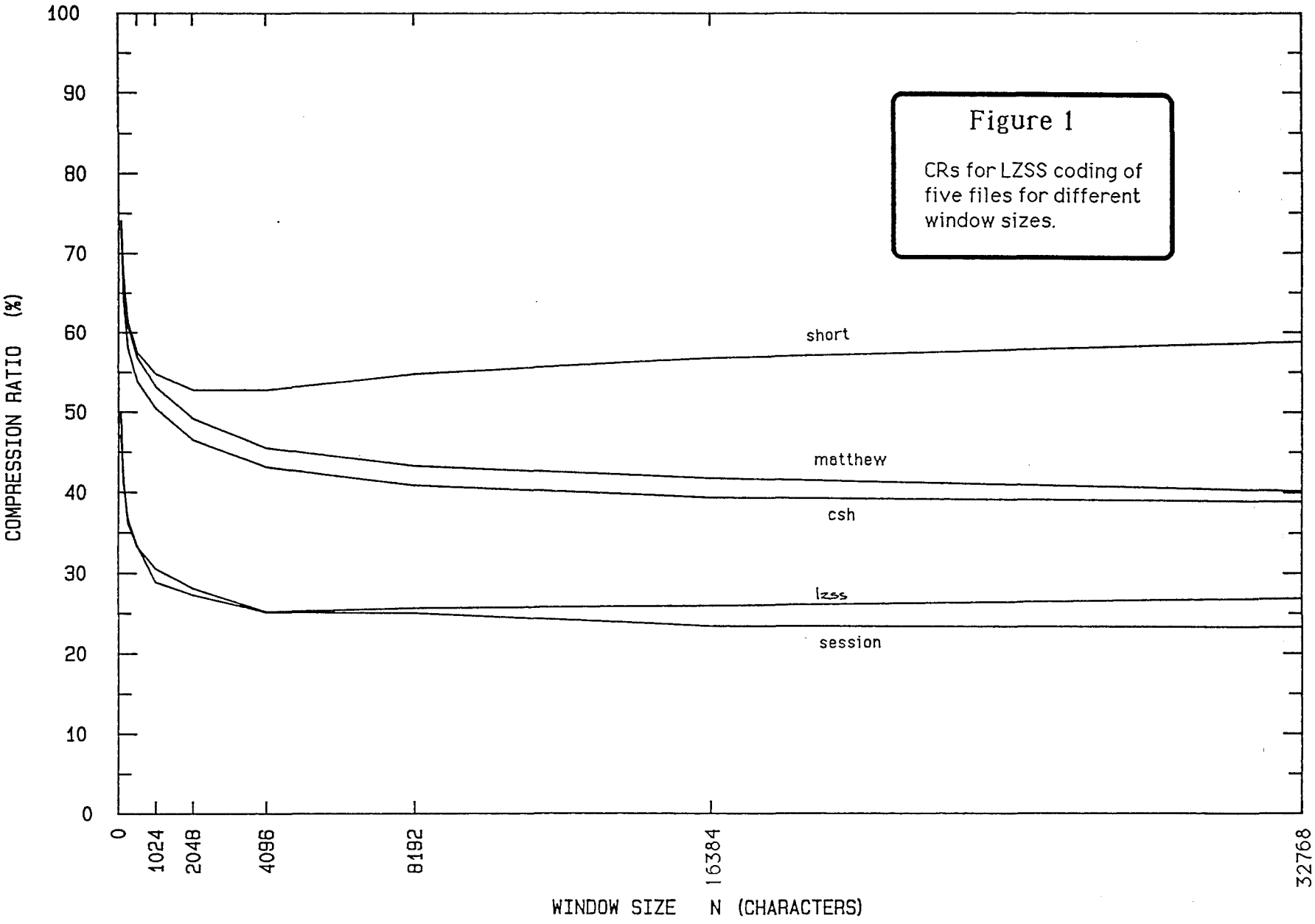
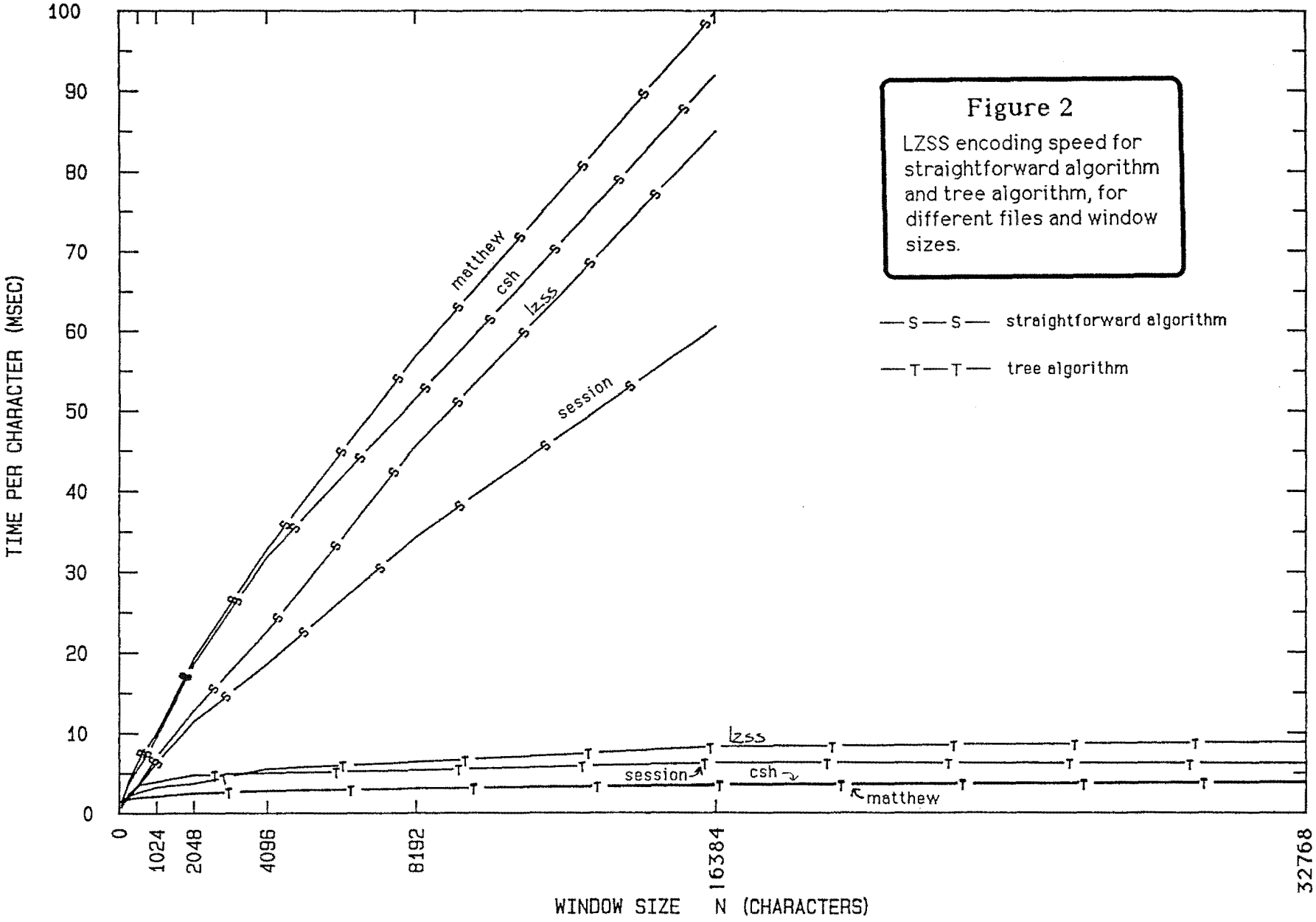


Figure 1

CRs for LZSS coding of five files for different window sizes.


TIME PER CHARACTER (MSEC)
100
90
80
70
60
50
40
30
20
10
0
matthew
csh
lzss
session
lzss
session
csh
matthew
0
1024
2048
4096
8192
16384
32768
WINDOW SIZE N (CHARACTERS)
Figure 2
LZSS encoding speed for straightforward algorithm and tree algorithm, for different files and window sizes.
S S straightforward algorithm
T T tree algorithm

Appendix B
The LZB text compression scheme

B.1 Description

LZB, a variation of the Ziv-Lempel sliding window greedy schemes, was developed in chapter 6. An example of LZB coding is given in section B.2.

LZB is parameterised by N, the size of the sliding window, and p, the minimum useful match length. N should be chosen to be a power of two, and LZB usually performs best if N is as large as possible. N=8192 performs well for a variety of texts. The value of p varies with N, and is typically $\lfloor (\log_2 N + 3)/8 \rfloor$. Some experimentation may be needed to find the optimal value of p, but again it is not critical. For N=8192, p=3 is optimal for all of the benchmark texts of section 2.2.

At each coding step, the N characters immediately prior to the unencoded part are available in a "sliding window". The sliding window is implemented as an array with character number *i* stored at location (*i* modulo N) in the array.

The window is searched to find the longest match for the characters to the right of the encoding position. The first character of the match must start in the window, but the match may extend into the unencoded characters. The match can be any length, but in practice it might be helpful to limit it to some value larger than any likely match length.

If the match length is *l*, and $l \geq p$, then the next *l* characters are coded as a pointer. Otherwise the next single character is transmitted explicitly. The characters just encoded are then added to the window, and the same number of characters automatically move out of the window to make room.

A flag bit precedes each character and pointer to distinguish between them. An explicit character is coded using a 7-bit ASCII code. A pointer has two components. The first is the location in the window of the match. If encoding is up to character i, then this component of the pointer can be transmitted in $\min(\lceil \log_2 i \rceil, \lceil \log_2 N \rceil)$ bits. The second component is the match length. This value has (p-1) subtracted so that it ranges from 1, and then is coded using the variable length coding of the integers C_γ, given in Appendix C. C_γ codes the value k in $2\lfloor \log_2 k \rfloor + 1$ bits.

The LZB decoder maintains a window in the same way as the encoder, but instead of searching for a match in the window it copies characters into the window using the pointers, or from the explicit characters.

B.2 Example

Figure B.1 shows how LZB coding works for the first 12 characters of the string "ababbabbbabcaa···", with N=8 and p=2. A pointer is represented as $<m,l>$, where m is the location of the match (coded in binary), and l is the adjusted length of the match (coded using C_γ).

Number of characters encoded	Input string and window	Match location	Match length	Output
0	|a b a b b a b b b a b c a a ···	?	0	a
1	[a] b a b b a b b b a b c a a ··· 0	?	0	b
2	[a b] a b b a b b b a b c a a ··· 0 1	0	2	<0,1>
4	[a b a b] b a b b b a b c a a ··· 0 1 2 3	1	4	<01,011>
8	[a b a b b a b b] b a b c a a ··· 0 1 2 3 4 5 6 7	1	3	<001,010>
11	a b a [b b a b b b a b] c a a ··· 3 4 5 6 7 0 1 2	?	0	c
12	a b a b [b a b b b a b c] a a ··· 4 5 6 7 0 1 2 3			

Figure B.1: LZB coding of the string "ababbabbbabcaa···".

Appendix C
Variable length codings of the integers

The common binary fixed-length coding of the integers requires a prespecified number of bits for each integer, say n. This coding is able to represent integers in the range 1 to 2^n. Sometimes the maximum value to be coded is not known and so a variable length coding of the integers must be used instead. The variable length codings here can code any value greater than zero.

For all the codes described here the length of the encoding increases with the value being encoded. Information theory tells us that if the coding of the integer i uses $|C(i)|$ bits then it is optimal if the probability of i is $2^{-|C(i)|}$. Thus the different codings given here imply different probability distributions, and the most appropriate coding for an application is the one with the probability distribution closest to that of the integers being encoded. It is possible to generate an optimal set of codes (using an integer number of bits) using Huffman's algorithm [Huffman 52], but this requires that a probability distribution is provided, and it is slower.

In the following, the definitions of C_α, C_β, $C_{\hat{\beta}}$, C_γ, $C_{\hat{\gamma}}$, C_δ and C_ω are due to Elias [Elias 75]. Examples of the codings are given in Table C.1.

C_α: C_α is unary coding; that is, $\alpha(i)$ is (i-1) zeros followed by a one, or $\alpha(1)=1$, $\alpha(i+1)= 0.a(i)$. $|a(i)|=i$.

C_β: C_β is binary coding; that is, $\beta(1)=1$, $\beta(2i)=\beta(i).0$, $\beta(2i+1)=\beta(i).1$. This coding is not decodable unless the length of the code is known in advance. Note that the most significant bit of $\beta(i)$ is always a one. $|\beta(i)| = \lfloor \log_2 i \rfloor + 1$.

$C_{\hat{\beta}}$: $C_{\hat{\beta}}$ is C_β with the most significant bit removed. It is not useful on its own, but is used to build other codes. $|\hat{\beta}(i)| = \lfloor \log_2 i \rfloor$.

C_γ: C_γ is a unary code for the number of bits in the binary coding of the integer, followed by the the binary coding of the integer with the most significant bit removed; that is, $\gamma(i) = \alpha(|\beta(i)|).\hat{\beta}(i)$. $|\gamma(i)| = 2\lfloor \log_2 i \rfloor + 1$. This coding was discovered independently by Bentley and Yao [Bentley 76].

$C_{\gamma'}$: $C_{\gamma'}$ can be viewed as $\lfloor \log_2 i \rfloor$ zeros followed by a one, and then $\hat{\beta}(i)$. $C_{\gamma'}$ is a rearrangement of C_γ, with each of the $\lfloor \log_2 i \rfloor$ zeros followed by a bit from $\hat{\beta}(i)$, and ending with a one. Thus, $|\gamma'(i)| = |\gamma(i)|$, but $C_{\gamma'}$ may be easier to implement than C_γ.

C_δ: C_δ is the C_γ coding of the length of the integer, followed by the integer in binary; that is, $\delta(i) = \gamma(|\beta(i)|)\hat{\beta}(i)$, and $|\delta(i)| = 1 + \lfloor \log_2 i \rfloor + 2\lfloor \log_2(1+\lfloor \log_2 i \rfloor) \rfloor$.

C_ω: With $\omega(i)$, first the code $\beta(i)$ is written. To the left of this the C_β representation of $(|\beta(i)|-1)$ is written. This process is repeated recursively, with each earlier code being the binary encoding of the length, less one, of the following code. The process halts on the left with a coding of 2 bits. A single zero is appended on the right to mark the end of the code.

RPE: RPE is a variable length coding used by Rodeh, Pratt and Even [Rodeh 81]. It is very similar to C_ω, but codes begin with 3 bits rather than 2, and the integer zero can be coded.

RPE': RPE' is RPE altered so that the integers range from 1 instead of 0, and with a redundant bit removed from the codes for the integers 1 to 4.

$binesc_n$: $Binesc_n$ is a class of binary codings with escapes: $binesc_1$, $binesc_2$, etc. The evaluation of $binesc_n(i)$ is a recursive procedure. If $i < 2^n$ then i is coded as an n-bit binary number. Otherwise the escape code, 0, is sent (using n bits), followed by $binesc_n(i-2^n+1)$. Table C.1 shows $binesc_2$.

$|binesc_n(i)| = n((i-1) \text{ div } (2^n-1) + 1)$. $Binesc_1$ is the same as C_α.

i	$\alpha(i)$	$\beta(i)$	$\hat{\beta}(i)$	$\gamma'(i)$	$\gamma(i)$	$\delta(i)$	$\omega(i)$	rpe(i)	rpe'(i)	$binesc_2(i)$
1	1	1		1:	1	1:	0	001:0	000:	01
2	01	10	0	01:0	001	001:0	10:0	010:0	001:	10
3	001	11	1	01:1	011	001:1	11:0	011:0	010:	11
4	0001	100	00	001:00	00001	011:00	10:100:0	100:0	011:	00:01
5	00001	101	01	001:01	00011	011:01	10:101:0	101:0	100:0	00:10
6	000001	110	10	001:10	01001	011:10	10:110:0	110:0	101:0	00:11
7	0000001	111	11	001:11	01011	011:11	10:111:0	111:0	110:0	00:00:01
8	00000001	1000	000	0001:000	0000001	00001:000	11:1000:0	100:1000:0	111:0	00:00:10
9	000000001	1001	001	0001:001	0000011	00001:001	11:1001:0	100:1001:0	100:1000:0	00:00:11
10	0000000001	1010	010	0001:010	0001001	00001:010	11:1010:0	100:1010:0	100:1001:0	00:00:00:01
11	00000000001	1011	011	0001:011	0001011	00001:011	11:1011:0	100:1011:0	100:1010:0	00:00:00:10
12	...	1100	100	0001:100	0100001	00001:100	11:1100:0	100:1100:0	100:1011:0	00:00:00:11
13		1101	101	0001:101	0100011	00001:101	11:1101:0	100:1101:0	100:1100:0	00:00:00:00:01
14		1110	110	0001:110	0101001	00001:110	11:1110:0	100:1110:0	100:1101:0	00:00:00:00:10
15		1111	111	0001:111	0101011	00001:111	11:1111:0	100:1111:0	100:1110:0	00:00:00:00:11
16		10000	0000	00001:0000	000000001	00011:0000	10:100:10000:0	101:10000:0	100:1111:0	...
17		10001	0001	00001:0001	000000011	00011:0001	10:100:10001:0	101:10001:0	101:10000:0	
18		10010	0010	00001:0010	000001001	00011:0010	10:100:10010:0	101:10010:0	101:10001:0	
19		10011	0011	00001:0011	000001011	00011:0011	10:100:10011:0	101:10011:0	101:10010:0	
20		10100	0100	00001:0100	000100001	00011:0100	10:100:10100:0	101:10100:0	101:10011:0	
21		10101	0101	00001:0101	000100011	00011:0101	10:100:10101:0	101:10101:0	101:10100:0	
22		10110	0110	00001:0110	000101001	00011:0110	10:100:10110:0	101:10110:0	101:10101:0	
23		10111	0111	00001:0111	000101011	00011:0111	10:100:10111:0	101:10111:0	101:10110:0	
24		11000	1000	00001:1000	010000001	00011:1000	10:100:11000:0	101:11000:0	101:10111:0	
25		11001	1001	00001:1001	010000011	00011:1001	10:100:11001:0	101:11001:0	101:11000:0	
26		11010	1010	00001:1010	010001001	00011:1010	10:100:11010:0	101:11010:0	101:11001:0	
27		11011	1011	00001:1011	010001011	00011:1011	10:100:11011:0	101:11011:0	101:11010:0	
28		11100	1100	00001:1100	010100001	00011:1100	10:100:11100:0	101:11100:0	101:11011:0	
29		11101	1101	00001:1101	010100011	00011:1101	10:100:11101:0	101:11101:0	101:11100:0	
30		11110	1110	00001:1110	010101001	00011:1110	10:100:11110:0	101:11110:0	101:11101:0	
31		11111	1111	00001:1111	010101011	00011:1111	10:100:11111:0	101:11111:0	101:11110:0	
32		100000	00000	000001:00000	00000000001	01001:00000	10:101:100000:0	110:100000:0	101:11111:0	

Table C.1: Examples of variable length codings of the integers.
Colons have been added for clarity.

Glossary

The glossary includes the names given to the TC schemes described in chapter 2.

The references are to sections in the thesis containing fuller descriptions.

adaptive coding — A class of compression schemes where the *model* used for coding is based on the text already encoded (1.2.4).

alphabet — The set of all possible *characters* which can occur in a text, usually denoted as A (1.4).

arithmetic coding — A class of *coder* which can assign codes for a given probability distribution with an average length arbitrarily close to the *entropy* (2.5).

bigram — A synonym for *digram.*

character — Any member of a character set such as ASCII (1.4).

coder — The part of an *encoder* which transmits text using probabilities generated by a *model* (1.4).

code space — The estimated probability of a symbol (1.4).

"compact" — A TC scheme (2.4.1). Note that compact performs *compression* rather than *compaction*!

compaction — Reducing the size of a file without removing any relevant information (1.2.3). See also *compression.*

"compress" — A TC scheme, also called *LZC* (2.6.9).

compression — *Compaction* which is completely reversible (1.2.3).

Compression Ratio — The size of a compressed file expressed as a percentage of the original file (2.2).

context — The symbols used by a *(Markov) model* on which a prediction is based.

CR — *Compression Ratio.*

DAFC	The "Double Adaptive File Compression" TC scheme (2.5.3).
decoder	An algorithm to perform decompression (1.2.1).
digram	A pair of letters (2.3.2).
digram coding	An approach to TC which uses *digrams* (2.3.2).
DMC	The "Dynamic Markov Compression" TC scheme (2.5.5).
encoder	An algorithm which performs *compression* (1.2.1).
entropy	The minimum number of bits per character that a text can be coded in for a given probability distribution [Shannon 51]. It is calculated as Σ_i-$P(x_i)\log_2 P(x_i)$, where $P(x_i)$ is the estimated probability of the symbol x_i.
FCA	Finite Context Automaton (3.2).
"first-order"	A TC scheme using a first-order *Markov model* (2.5.2).
Greedy Macro	A *macro encoding* scheme where strings are matched with the text using a greedy algorithm (5.1.1).
GM	*Greedy Macro.*
Huffman coding	An algorithm for assigning variable length codes for a given probability distribution. It is a form of *arithmetic coding*, but is optimal only if the probabilities are negative powers of two (2.4).
LZ	Ziv-Lempel coding (2.6). The following schemes are forms of LZ coding.
LZ76	(2.6.1)
LZ77	(2.6.4)
LZ78	(2.6.7)
LZB	(Appendix B)
LZC	(2.6.9) - also known as *"compress"*.
LZJ	(2.6.10)
LZR1	(2.6.2)
LZR2	(2.6.3)
LZR3	(2.6.5)
LZSS	(2.6.6)
LZW	(2.6.8)
macro encoding	TC schemes where substrings of the text are replaced with codes. Some sort of dictionary of substrings is used to assign the codes (5.1).

"Macwrite" The TC scheme used by the Macintosh word processor, Macwrite (2.3.1).

Markov model A *model* where each *symbol* is *predicted* by a finite number of immediately preceding symbols (*context*). The *order* of a Markov model is the number of symbols used as the context (2.5.2).

model An approximation to the process of generating text. A model is used to *predict* text (1.4).

MTF The "Move To Front" TC scheme (2.5.6)

OPM/L Original Pointer Macro coding/Left pointers. That is, ***macro encoding*** where the substrings are chosen from the **original** text to the **left** of the encoding position, and replaced with a **pointer** to the substring [Storer 82].

order The order of a *Markov model* is the number of symbols used as a context to predict the next symbol (2.5.2).

overall compression
The weighted average of the CRs for the six benchmark texts, for a given TC scheme (2.2).

"pack" A TC scheme which uses simple *Huffman coding* (2.4.1).

Pike's scheme A TC scheme (2.3.3).

PPM The "Prediction by Partial Matching" TC scheme (2.5.4).

prediction Assigning probabilities to forthcoming *symbols* (or events).

prefix The string $s=s_1\cdots s_n$ is a prefix of the string $t=t_1\cdots t_m$ iff $n\leq m$ and $s_1=t_1$, $s_2=t_2 \cdots s_n=t_n$.

proper prefix The string s is a proper prefix of t iff s is a *prefix* of t and |s| is strictly less than |t|.

scheme A compression scheme is a pair of algorithms which perform *compression* and *decompression* (1.2.1).

string A finite sequence of *characters* (1.4).

symbol An easily recognised string, such as a character or word (1.4).

symbol-wise — A model which *predicts* each symbol of the text being encoded (5.4).

text — Data which is represented using a standard character set such as ASCII (1.2.2).

TC — Text Compression (1.2.1).

trie — A data structure which enables fast searching for strings and substrings. A trie is a multiway tree with a path for each string inserted in it [Wirth 76].

variable-order Markov model (VOMM) — A *Markov model* where the *order* varies in an attempt to obtain good prediction (3.1).

"zero-order" — A TC scheme which uses a zero-order *Markov model* (2.5.2).

Ziv-Lempel — A class of TC schemes based on work done by Jacob Ziv and Abraham Lempel [Lempel 76, Ziv 77, Ziv 78]. See also LZ76, LZ77, LZ78, LZR1, LZR2, LZR3, LZSS, LZW, LZC, LZJ, and LZB (2.6).

Acknowledgements

I wish to record my thanks to the many people who have helped with this research. Prof. John Penny has patiently supervised the work and has been a valued source of good advice and encouragement. Profs. John Cleary and Ian Witten have been most helpful in suggesting directions for research. The equivalence of LZ and DMC coding to Markov models was originally proposed by John Cleary.

The staff of the Computer Science department at Canterbury have provided unfailing support. In particular, Bruce McKenzie and Krzysztof Pawlikowski have given helpful comments for several reports and papers. Alistair Moffat has provided much stimulating discussion, and is responsible for some refinement of the proofs in chapter 4. He has also made available results from experiments with the MTF scheme, and the benchmark text "zen".

Nigel Horspool and Gordon Cormack provided helpful comments, and made available the DMC program and early versions of [Cormack 87].

Financial support was provided by post-graduate scholarships from the University Grants Committee and from IBM (NZ).

Judith, my wife, has offered much support during this work, and has typed much of the thesis.

Finally, I would like to express a deep gratitude to family and friends who have supported me with their love and prayers, and for the faithfulness of God, who initiated this project, and has guided it to this stage.

References

[Bassiouni 85]MA Bassiouni, Data compression in scientific and statistical databases, *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 10, pp. 1047-1058, Oct. 1985.

[Bell 86] TC Bell, Better OPM/L Text Compression, *IEEE Trans. Comm.*, to appear, December 1986.

[Bentley 76] JL Bentley and AC Yao, An almost optimal algorithm for unbounded searching, *Inf. Process. Lett.*, vol. 5, no. 3, pp. 82-87, Aug. 1976.

[Bentley 86] JL Bentley, DD Sleator, RE Tarjan and VK Wei, A locally adaptive data compression scheme, *C.ACM*, vol. 29, no. 4, pp. 320-330, April 1986.

[Bookstein 76]A Bookstein and G Fouty, A mathematical model for estimating the effectiveness of bigram coding, *Inf. Proc. Manag.*, vol. 12, pp. 111-116, 1976.

[Bourne 61] CP Bourne and DF Ford, A study of methods for systematically abbreviating English words and names, *J. ACM*, vol. 8, pp. 538-552, 1961.

[Cleary 84a] JG Cleary and IH Witten, A comparison of Enumerative and Adaptive Codes, *IEEE Trans. Inf. Th.*, vol. IT-30, no. 2, p. 306, 1984.

[Cleary 84b] JG Cleary and IH Witten, Data compression using adaptive coding and partial string matching, *IEEE Trans. Comm.*, vol. COM-32, no. 4, pp. 396-402, July 1984.

[Cormack 84] GV Cormack and RN Horspool, Algorithms for adaptive Huffman codes, *Inf. Process. Lett.*, vol. 18, no. 3, pp. 159-165, March 1984.

[Cormack 87] GV Cormack and RN Horspool, Data compression using Dynamic Markov Modelling, *Computer Journal*, to appear, 1987.

[Elias 75] P Elias, Universal codeword sets and representations of the integers, *IEEE Trans. Inf. Th.* , vol. IT-21, no. 2, pp. 194-203, March 1975.

[Even 78] S Even and M Rodeh, Economical encoding of commas between strings, *C.ACM*, vol. 21, pp. 315-317, April 1978.

[Fair 86] EE Fair, Backbone automatic news - compression, received on *net.news* and *net.micro.mac*, 22 Sept., 1986.

[Gallager 78] RG Gallager, Variations on a theme by Huffman, *IEEE Trans. Inf. Th.*, IT-24, no. 6, pp. 668-674, 1978.

[Gilbert 59] EN Gilbert and EF Moore, Variable length binary encoding, *Bell System Technical Journal*, pp. 933, July 1959.

[Gonzalez 85] ME Gonzalez-Smith and JA Storer, Parallel Algorithms for Data Compression, *J. ACM*, vol. 32, no. 2, pp. 344-373, April 1985.

[Gottlieb 75] D Gottlieb, SA Hagerth, PGH Lehot and HS Rabinowitz, A classification of compression methods and their usefulness for a large data processing centre, *National Comp. Conference*, vol. 44, pp. 453-458, 1975.

[Guazzo 80] M Guazzo, A general minimum redundancy source coding algorithm, *IEEE Trans . Inf. Th.*, vol. 26, pp. 15-25, 1980.

[Hahn 74] B Hahn, A new technique for compression and storage of data, *C.ACM*, vol. 17, no. 8, pp. 434-436, August 1974.

[Horspool 86] RN Horspool and GV Cormack, Technical correspondence, *C.ACM*, vol. 29, no. 2, pp. 150-152, Feb. 1986.

[Hu 71] Hu and Tucker, Optimal computer search trees and variable length alphabetical codes, *SIAM Journal of Applied Mathematics*, vol. 21, p. 514, 1971.

[Huffman 52] DA Huffman, A method for the construction of minimum redundancy codes, *Proc. Institute of Radio Engineers*, vol. 40, pp. 1098-1101,1952.

[Jakobsson 85] M Jakobsson, Compression of character strings by an adaptive dictionary, *BIT*, vol. 25, pp. 593-603, 1985.

[Jewell 76] GC Jewell, Text compaction for information retrieval systems, *IEEE Systems, Man and Cybernetics Soc. Newsletter*, vol. 5, p. 47, 1976.

[Jones 81] CB Jones, An efficient coding system for long source sequences, *IEEE Trans Inf. Th.*, vol IT-27, pp. 280-291, May 1981.

[Kain 72] RY Kain, Automata Theory: Machines and Languages, McGraw-Hill, New York, 1972.

[Knuth 85] DE Knuth, Dynamic Huffman coding, *J.Alg.*, vol. 6, pp. 163-180, 1985.

[Lambert 86] S Lambert and S Ropvequets (editors), CD ROM: The new papyrus, Microsoft Press, pp. 234-235, 1986.

[Langdon 81] GG Langdon, Tutorial on arithmetic coding, IBM research report no. RJ3128 (38505) 5/6/81, 1981.

[Langdon 83a] G Langdon, A note on the Ziv-Lempel model for compressing individual sequences, *IEEE Trans. Inf. Th.*, vol. IT-29, pp. 284-287, 1983.

[Langdon 83b] G Langdon and J Rissanen, A double-adaptive file compression algorithm, *IEEE Trans. Comm.*, vol. COM-31, no. 11, pp. 1253-1255, 1983.

[Lea 78] RM Lea, Text compression with an Associative Parallel Processor, *Computer Journal*, vol. 21, no. 1, pp. 45-56, 1978.

[Lempel 76] A Lempel and J Ziv, On the complexity of finite sequences, *IEEE Trans. Inf. Th.*, vol. IT-22, no. 1, pp. 75-81, Jan. 1976.

[Lynch 73] MF Lynch, Compression of bibliographic files using an adaptation of run-length coding, *Inf. Stor. Retr.*, vol. 9, pp. 207-214, 1973.

[Mayne 75] A Mayne and EB James, Information compression by factorising common strings, *Computer Journal* , vol. 18, no. 2, pp. 157-160, 1975.

[McCreight 76] EM McCreight, A space-economical suffix tree construction algorithm, *J. ACM*, vol. 23, no. 2, pp. 262-272, April 1976.

[McIntyre 85] DR McIntyre and MA Pechura, Data compression using static Huffman code-decode tables, *C. ACM*, vol. 28, no. 6, pp. 612-616, June 1985.

[Money 79] SA Money, Teletext and Viewdata, Butterworth & Co., London, 1979

[Ozeki 74a] K Ozeki, Optimal encoding of linguistic information, Systems·Computers·Controls, vol. 5, no. 3, pp. 96-103, 1974.

[Ozeki 74b] K Ozeki, Stochastic context-free grammar and Markov chain, Systems·Computers·Controls, vol. 5, no. 3, pp. 104-110, 1974.

[Ozeki 75] K Ozeki, Encoding of linguistic information generated by a Markov chain which is associated with a stochastic context-free grammar, Systems·Computers·Controls, vol. 6, no. 3, pp. 75-80, 1975.

[Pascoe 76] R Pasco, Source coding algorithms for fast data compression, PhD Thesis, Stanford University, 1976.

[Pike 81] J Pike, Text compression using a 4-bit coding system, *Computer Journal*, vol. 24, no.4 , pp. 324-330, 1981.

[Rissanen 76] Generalized Kraft inequality and arithmetic coding, *IBM J. Research and Development*, vol. 20, pp. 198-203, May 1976.

[Rissanen 79] J Rissanen and G Langdon, Arithmetic Coding, *IBM J. Res. Develop.*, vol. 23, no. 2, pp. 149-162, March 1979.

[Rissanen 81] J Rissanen and G Langdon, Universal modeling and coding, *IEEE Trans. Inf. Th.*, vol. IT-27, pp. 12-23,1979.

[Roberts 82] MG Roberts, Local order estimating Markovian analysis for noiseless source coding and authorship identification, PhD Thesis, Stanford University, June 1982.

[Rodeh 81] M Rodeh, VR Pratt and S Even, Linear algorithm for data compression via string matching, *J. ACM*, vol. 28, no. 1, pp. 16-24, 1981.

[Rubin 76] F Rubin, Experiments in text file compression, *C.ACM*, vol. 19, no. 11, pp. 617-623, 1976.

[Ruth 72] Ruth and Kreutzer, Data compression for large business files, *Datamation*, pp. 62-66, Sept. 1972.

[Schieber 71] WD Schieber and GW Thomas, An algorithm for compaction of alphanumeric data, *J. Library Automation*, vol. 4, pp. 198-206, 1971.

[Schuegraf 73] EJ Schuegraf and Heaps, Selection of equifrequent word fragments for information retrieval, *Info.Stor. Retr.*, vol. 9, pp. 697-711, 1973.

[Schuegraf 74] EJ Schuegraf and Heaps, A comparison of algorithms for database compression by use of fragments as language elements, *Info. Stor. Retr.*, vol. 10, pp. 309-319, 1974.

[Schwartz 64] ES Schwartz and B Kallick, Generating an economical prefix code, *C. ACM*, vol. 7, pp. 166-169, 1964.

[Severance 83] DG Severance, A practitioner's guide to database compression, *Inf. Syst.*, vol. 8, no. 1, pp. 51-62, 1983.

[Shannon 48] CE Shannon, A mathematical theory of communication, *Bell System Technical Journal* , vol. 27, pp. 379-423, 623-656, 1948.

[Shannon 51] CE Shannon, Prediction and entropy of printed English., *Bell System Technical Journal*, vol. 30, no. 1, p. 50, 1951.

[Snyderman 70] M Snyderman and B Hunt, The myriad virtues of text compaction, *Datamation*, no. 1, pp. 36-40, Dec. 1970.

[Storer 82] JA Storer and TG Szymanski, Data compression via textual substitution, *J. ACM*, vol. 29, no.4, pp. 928-951, 1982.

[Thomas 85] SW Thomas and JW Orost, "compress" (version 4.0) program and documentation, available from joe@petsd.UUCP, 1985.

[Urrows 84] H Urrows and E Urrows, LaserData, Mnemos and other data disks: the race to store and retrieve with optics, *Videodisc and Optical Disk* , vol. 4, no. 2, pp. 130, March-April 1984.

[Wagner 73] RA Wagner, Common phrases and minimum-space text storage, *C. ACM*, vol. 16, no. 3, pp. 148-152, 1973.

[Welch 84] TA Welch, A technique for high performance data compression, *Computer*, vol. 17, no. 6, pp. 8-19, 1984.

[Weyer 85] SA Weyer and AH Borning, A prototype electronic encyclopaedia, *ACM Trans. on Office Information Systems*, vol. 3, no. 1, pp. 63-88, Jan. 1985.

[White 67] HE White, Printed English compression by dictionary encoding., *Proceedings of the IEEE*, vol. 55, no. 3, pp. 390-396, 1967.

[Wilson 76] G Wilson, The Old Telegraphs, Phillimore and Co., Chichester, England, 1976.

[Wirth 76] N Wirth, Algorithms + Data Structures = Programs, Prentice-Hall inc., Englewood Cliffs, N.J., pp. 201-211, 215-226, 1976.

[Witten 85] IH Witten and JG Cleary, Foretelling the future by adaptive modeling, Dept. of Comp. Sc., University of Calgary, March 1985.

[Witten 86] IH Witten, R Neal and JG Cleary, Arithmetic coding for data compression, University of Calgary Res. Rep. 86/238/12, August 1986.

[Wolff 78] JG Wolff, Recoding of natural language for economy of transmission or storage, *Computer Journal*, vol. 21, no. 1, pp. 42-44, 1978.

[Young 80] TY Young and PS Liu, Overhead storage considerations and a multilinear method for data file compression, *IEEE Trans Software Eng.*, vol. 6, no. 4, pp. 340-347, 1980.

[Young 85] DM Young, Macwrite file formats, *Wheels for the Mind*, Fall 1985.

[Zipf 49] GK Zipf, Human Behaviour and the Principle of Least Effort, Addison-Wesley, Reading, Mass., 1949.

[Ziv 77] J Ziv and A Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Inf. Th.*, vol. IT-23, no. 3, pp. 337-343, 1977.

[Ziv 78] J Ziv and A Lempel, Compression of individual sequences via variable-rate coding, *IEEE Trans. Inf. Th.*, vol. IT-24, no.5, pp. 530-536, 1978.

Postscript

The LZB scheme achieved a CR of 39.6% compressing the raw text of this thesis, not including this postscript, reducing the disk space used from 212 kbytes to 84 kbytes.